Part II
Processes and Threads
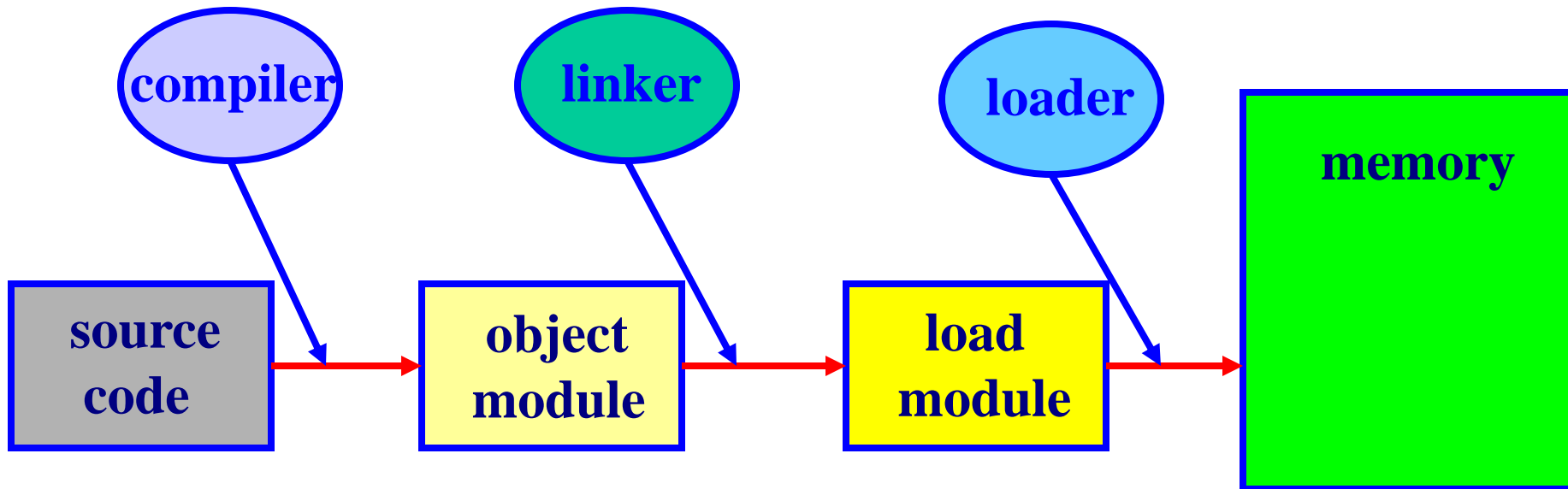## Process Basics

Program testing can be used to show the presence of bugs,
but never to show their absence

Spring 2015

Edsger W. Dijkstra
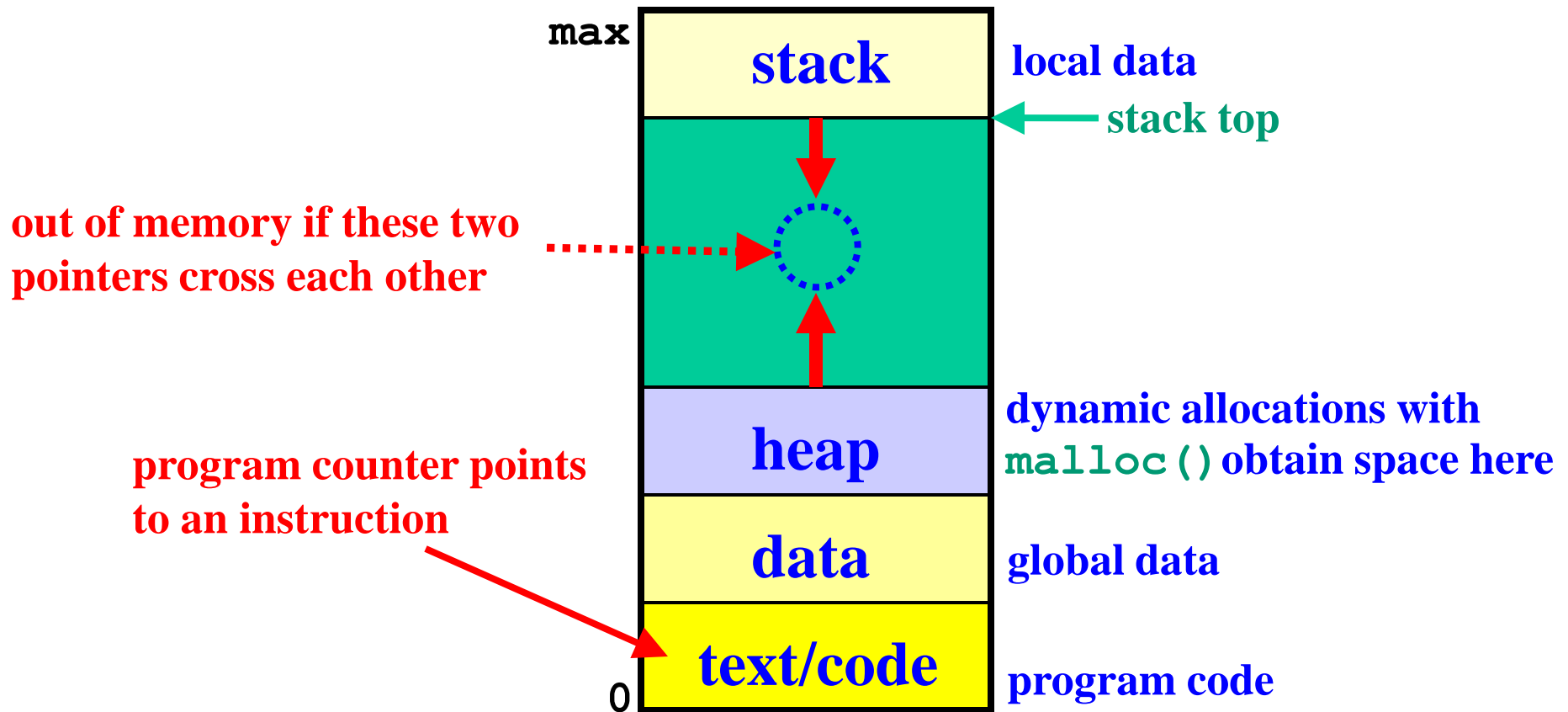
From Compilation to Execution

- **A compiler compiles source files to `.o` files.**
- **A linker links `.o` files and other libraries together, producing a binary executable (e.g., `a.out`).**
- **A loader loads a binary executable into memory for execution.**

What Is a Process?

- **When the OS runs a program (i.e., a binary executable), this program is loaded into memory and the control is transferred to this program's first instruction. Then, the program runs.**

- **A process is a program in execution.**

- **A process is more than a program, because a process has a program counter, stack, data section, code section, etc (i.e., the runtime stuffs).**

- **Moreover, multiple processes may be associated with one program (e.g., run the same program, say `a.out`, multiple times at the same time).**
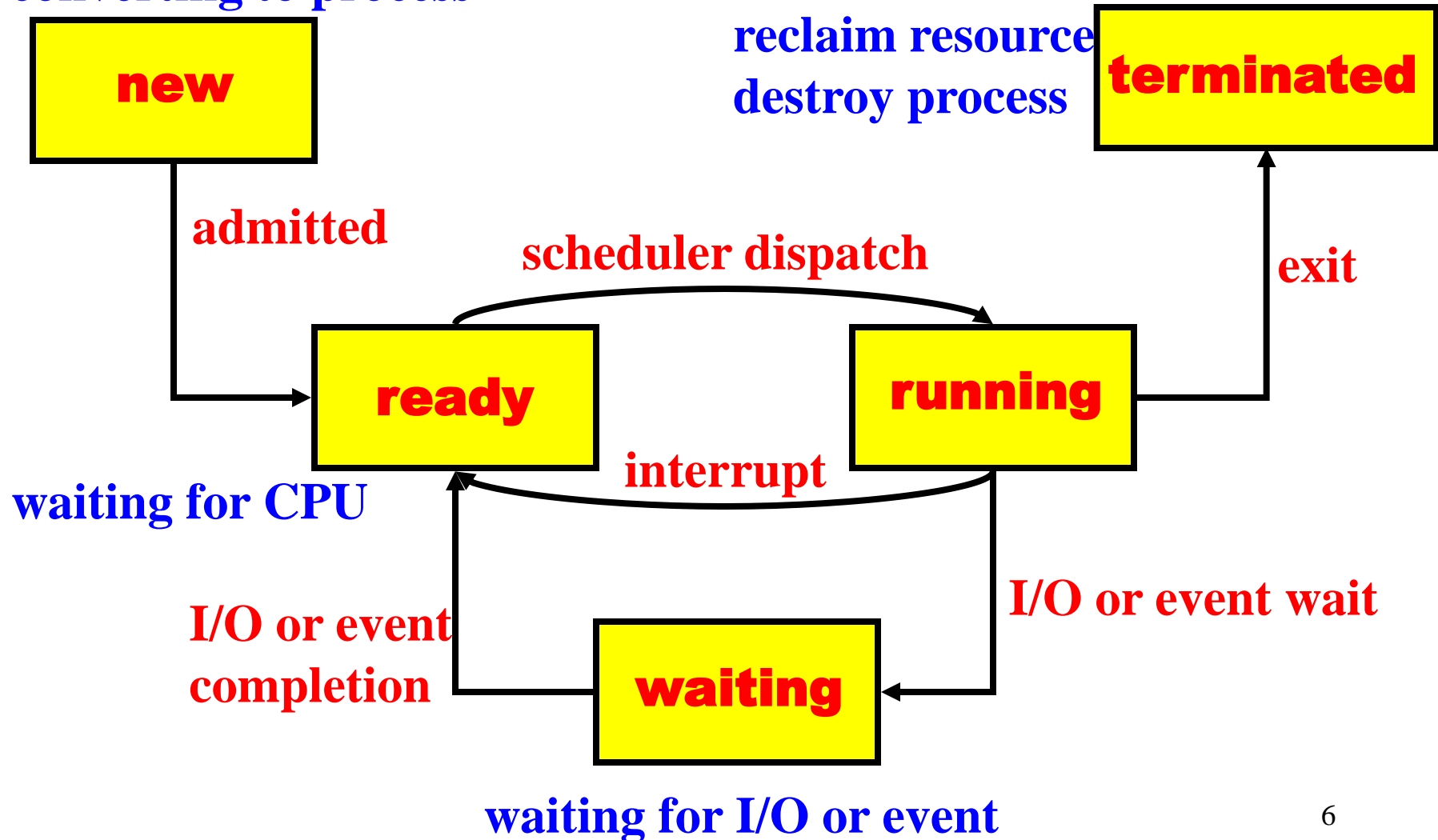
Process Space

max

stack — local data

← stack top

out of memory if these two pointers cross each other

heap — dynamic allocations with `malloc()` obtain space here

program counter points to an instruction

data — global data

text/code — program code

0

Process States

At any moment, a process can be in one of the five states: new, running, waiting, ready and terminated.

- **New**: The process is being created
- **Running**: The process is executing on a CPU
- **Waiting**: The process is waiting for some event to occur (e.g., waiting for I/O completion)
- **Ready**: The process has everything but the CPU. It is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.

Process State Diagram

converting to process

new

reclaim resource
destroy process

terminated

admitted

scheduler dispatch

exit

ready

running

interrupt

waiting for CPU

I/O or event
completion

I/O or event wait

waiting

waiting for I/O or event

Process Representation in OS

pointer	process state
process ID	
program counter	
registers	
scheduling info	
memory limits	
list of open files	
:	

- **Each process is assigned a unique number, the process ID.**
- **Process info are stored in a table, the process control block (PCB).**
- **These PCBs are chained into a number of lists. For example, all processes in the ready state are in the ready queue.**

Process Scheduling: 1/2

- **Since the number of processes may be larger than the number of available CPUs, the OS must maintain maximum CPU utilization.**

- **To determine which process can do what, processes are chained into a number of scheduling queues.**

- **For example, in addition to the ready queue, each event may have its own scheduling queue (*i.e.*, waiting queue).**
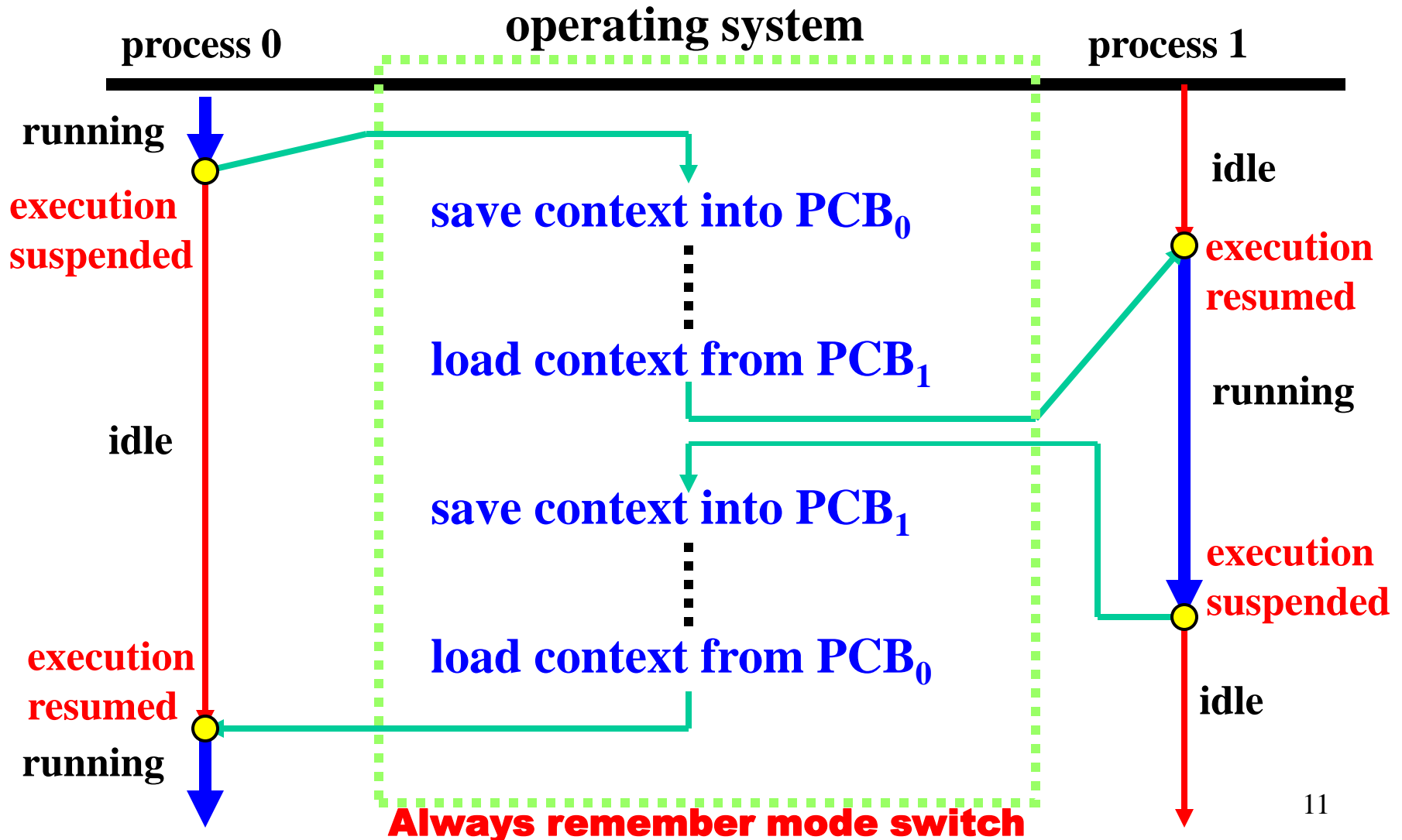
Process Scheduling: 2/2

- **The ready queue, which may be organized into several sub-queues, has all processes ready to run.**

- **The OS has a CPU scheduler.**

- **When a CPU is free, the CPU scheduler looks at the ready queue, picks a process, and resumes it.**

- **The way of picking a process from the ready queue is referred to as scheduling policy.**

- **Scheduling policy is unimportant to this course because we cannot make any assumption about it.**

Context Switch: 1/2

- **What is a process context?** **The context of a process includes process ID, process state, the values of CPU registers, the program counter, and other memory/file management information (i.e., execution environment).**

- **What is a context switch?** **After the CPU scheduler selects a process and before allocates CPU to it, the CPU scheduler must**
  - ➤ save the context of the currently running process,
  - ➤ put it back to the ready queue,
  - ➤ load the context of the selected process, and
  - ➤ go to the saved program counter.

Context Switch: 2/2

operating system

process 0

process 1

running

idle

execution suspended

save context into PCB_0

execution resumed

load context from PCB_1

running

idle

save context into PCB_1

execution suspended

execution resumed

load context from PCB_0

idle

running

Always remember mode switch

Operations on Processes

- **There are three commonly seen operations:**
  - ❖**Process Creation**: Create a new process. The newly created is the child of the original. Unix uses `fork()` to create new processes.

  - ❖**Process Termination**: Terminate the execution of a process. Unix uses `exit()`.

  - ❖**Process Join**: Wait for the completion of a child process. Unix uses `wait()`.

- `fork()`, `exit()` and `wait()` are system calls.

Some Required Header Files

- **Before you use processes, include header files `sys/types.h` and `unistd.h`.**

- **`sys/types.h` has all system data types, and `unistd.h` declares standard symbolic constants and types.**

```
#include  <sys/types.h>
#include  <unistd.h>
```

The fork() System Call

- **The purpose of `fork()` is to create a child process. The creating and created processes are the parent and child, respectively.**

- **`fork()` does not require any argument!**

- **If the call to `fork()` is successful, Unix creates an identical but separate address space for the child process to run.**

- **Both processes start running with the instruction following the `fork()` system call.**

fork() Return Values

- **A negative value means the creation of a child process was unsuccessful.**

- **A zero means the process is a child.**

- **Otherwise, `fork()` returns the process ID of the child process. The ID is of type `pid_t`.**

- **Function `getpid()` returns the process ID of the caller.**

- **Function `getppid()` returns the parent's process ID. If the calling process has no parent, `getppid()` returns `1`.**
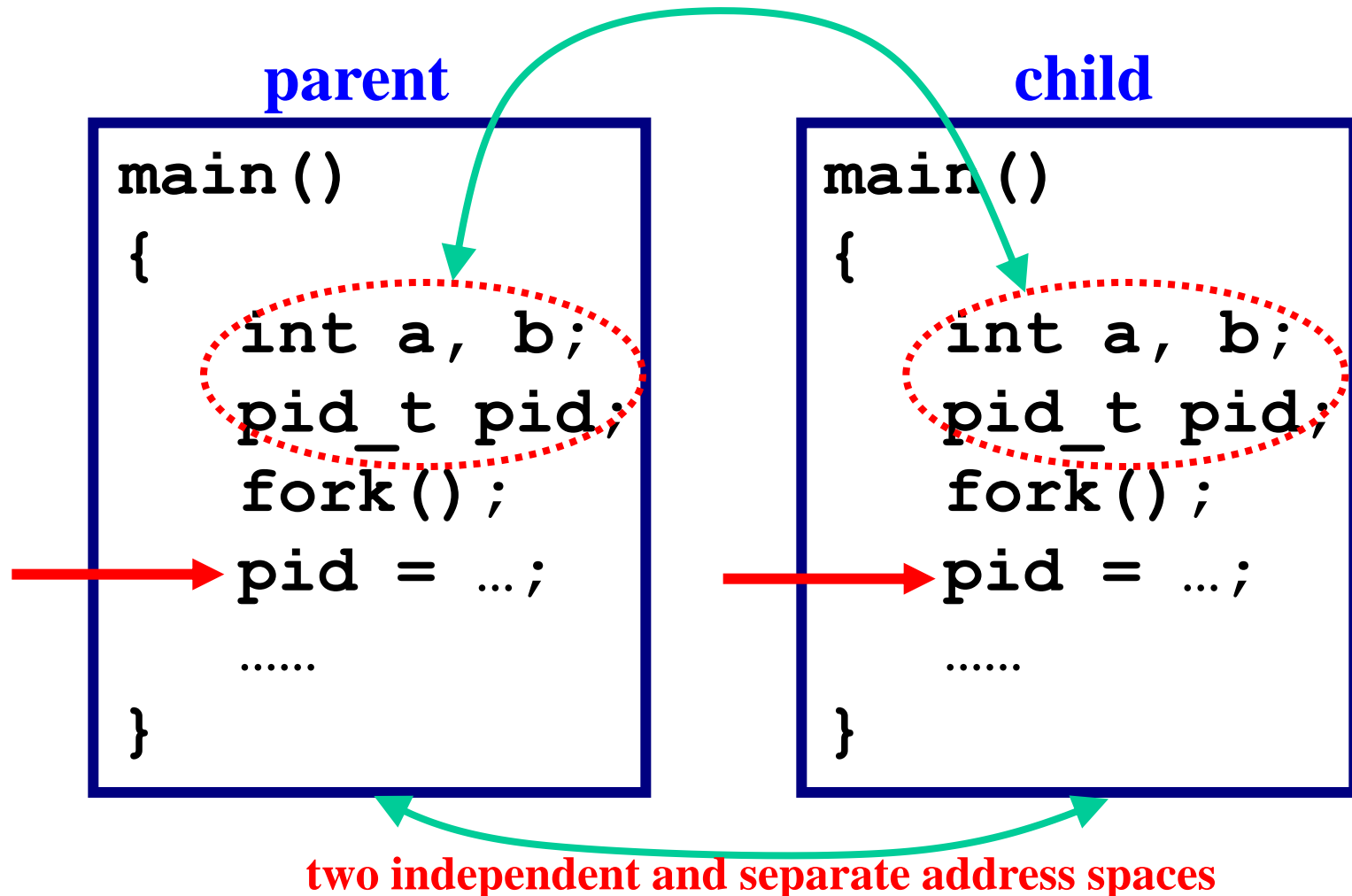
Before the Execution of fork()

parent

```
main()
{
    int a, b;
    pid_t pid
    fork();
    pid = …;
    ……
}
```

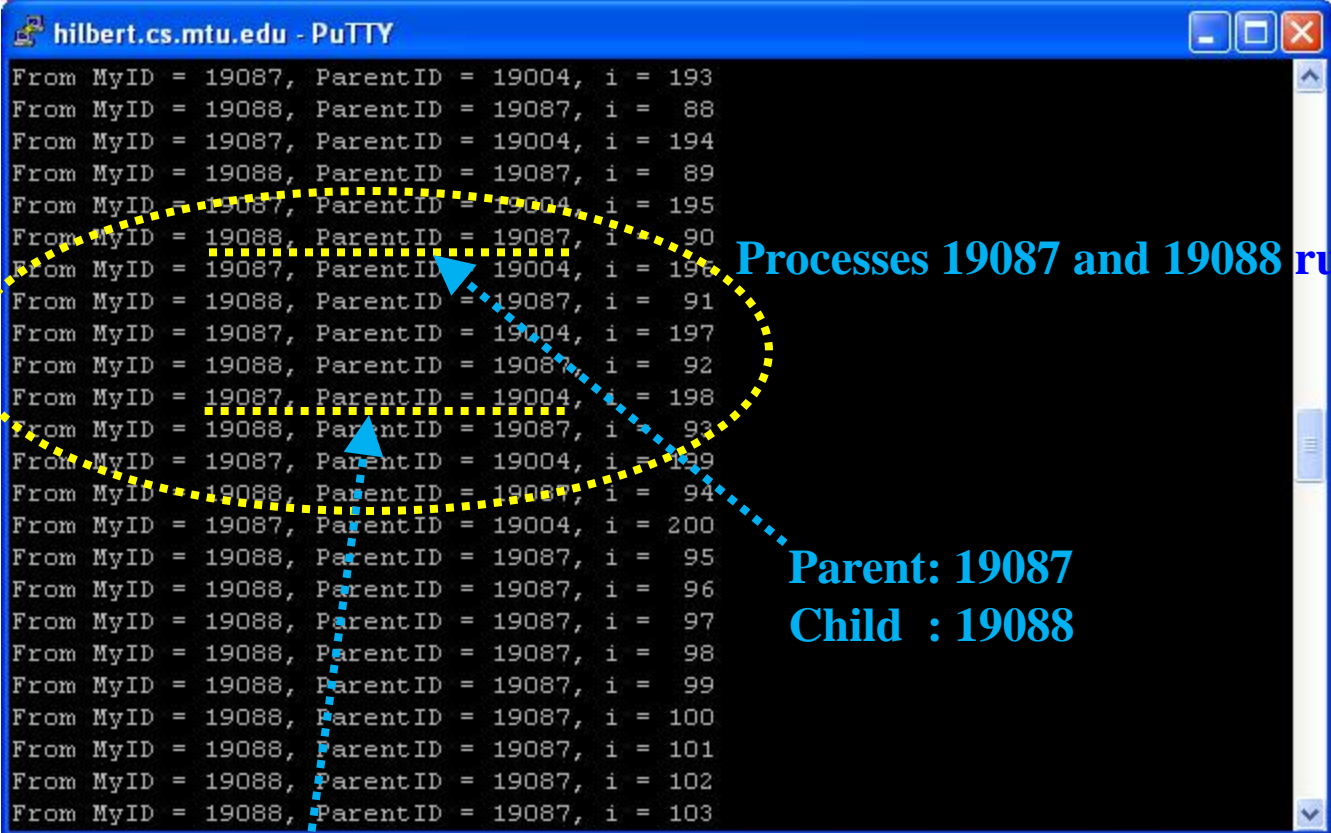# After the Execution of fork()

in different address spaces

parent   **child**

```
main()                        main()
{                             {
    int a, b;                     int a, b;
    pid_t pid;                    pid_t pid;
    fork();                       fork();
→   pid = …;                  →   pid = …;
    ……                            ……
}                             }
```

two independent and separate address spaces

Example 1: 1/2

```c
#include  <stdio.h>                          fork-1.c
#include  <string.h>
#include  <sys/types.h>
#include  <unistd.h>

void main(void)
{
   pid_t  MyID, ParentID;
   int    i;
   char   buf[100];

   fork();                      // create a child process
   MyID     = getpid();  // get my process ID
   ParentID = getppid(); // get my parent's process ID
   for (i = 1; i <= 200; i++) {
      sprintf(buf, "From MyID=%ld, ParentID=%ld, i=%3d\n",
                 MyID, ParentID, i);
      write(1, buf, strlen(buf));  // why don't we
   }                                 //     use printf?
}
```

this is `stdout`

Example 1: 2/2


```
hilbert.cs.mtu.edu - PuTTY
From MyID = 19087, ParentID = 19004, i = 193
From MyID = 19088, ParentID = 19087, i =  88
From MyID = 19087, ParentID = 19004, i = 194
From MyID = 19088, ParentID = 19087, i =  89
From MyID = 19087, ParentID = 19004, i = 195
From MyID = 19088, ParentID = 19087, i =  90
From MyID = 19087, ParentID = 19004, i = 196
From MyID = 19088, ParentID = 19087, i =  91
From MyID = 19087, ParentID = 19004, i = 197
From MyID = 19088, ParentID = 19087, i =  92
From MyID = 19087, ParentID = 19004, i = 198
From MyID = 19088, ParentID = 19087, i =  93
From MyID = 19087, ParentID = 19004, i = 199
From MyID = 19088, ParentID = 19087, i =  94
From MyID = 19087, ParentID = 19004, i = 200
From MyID = 19088, ParentID = 19087, i =  95
From MyID = 19088, ParentID = 19087, i =  96
From MyID = 19088, ParentID = 19087, i =  97
From MyID = 19088, ParentID = 19087, i =  98
From MyID = 19088, ParentID = 19087, i =  99
From MyID = 19088, ParentID = 19087, i = 100
From MyID = 19088, ParentID = 19087, i = 101
From MyID = 19088, ParentID = 19087, i = 102
From MyID = 19088, ParentID = 19087, i = 103
```

Processes 19087 and 19088 run concurrently

Parent: 19087
Child : 19088

19004
↓
19087
↓
19088

Parent 19087's parent is 19004, the shell that executes `fork-1`

19

fork(): A Typical Use

```
main(void)
{
  pid_t pid;

  pid = fork();
  if (pid < 0)
    printf("Oops!");
  else if (pid == 0)
    child();
  else // pid > 0
    parent();
}
```

```
void child(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf(" Child:%d\n", i);
  printf("Child done\n");
}

void parent(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf("Parent:%d\n", i);
  printf("Parent done\n");
}
```

we use `printfs` here to save space.
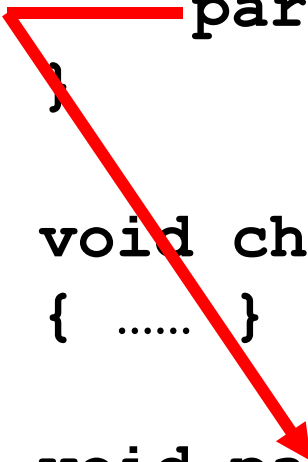
20

Before the Execution of fork()

parent

```
main(void)          pid=?
{
    pid = fork();
    if (pid == 0)
        child();
    else
        parent();
}

void child(void)
{ ...... }

void parent(void)
{ ...... }
```

After the Execution of fork() 1/2

parent

child

```
main(void)
{
  pid = fork();
→ if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```

pid=123

```
main(void)
{
  pid = fork();
→ if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```

pid = 0

in two different address spaces

parent

child

```
main(void)          pid=123
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```
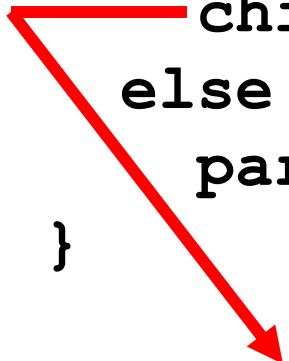
```
main(void)          pid = 0
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ …… }

void parent(void)
{ …… }
```

Example 2: 1/2

```
#include ………              // child exits first          fork-2.c
void main(void)
{
    pid_t  pid;
                        force the child to print after the parent terminates
    pid = fork();
    if (pid == 0) {              // child here
        printf("    From child %ld: my parent is %ld\n",
                        getpid(), getppid());
        sleep(5);
        printf("    From child %ld 5 sec later: parent %ld\n",
                        getpid(), getppid());
    }
    else {                       // parent here
        sleep(2);
        printf("From parent %ld: child %ld, parent %ld\n",
                    getpid(), pid, getppid());
        printf("From parent: done!\n");
    }
}
```
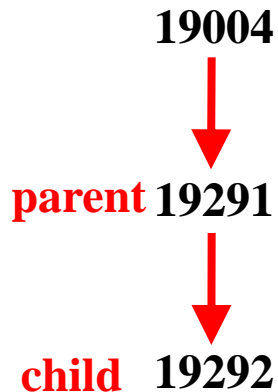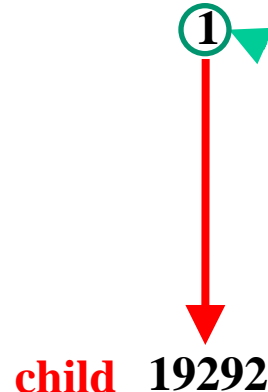
Example 2: 2/2


```
hilbert.cs.mtu.edu - PuTTY
HILBERT:/home/csdept/shene/CS3331/O3-Process(41) fork-2
    From child 19292: my parent is 19291
From parent 19291: my child is 19292, my parent is 19004
From parent: done!
    From child 19292 5 sec later: my parent is 1
```

Orphan children have a new parent `init`, the Unix process that spawns all processes

while parent is still running

19004

parent 19291

child 19292

parent terminated

①

1 is the ID of `init`, the Unix process that spawns all processes

child 19292

19004 is the shell process that executes 19291

25

Example 3: 1/2

fork-3.c

```
#include  ……    // separate address spaces

                              this is equivalent to
void  main(void)              pid = fork();
{                             if (pid == 0) …
   pid_t  pid;
   char   out[100];
   int    i = 10, j = 20;

   if ((pid = fork()) == 0) {   // child here
      i = 1000; j = 2000;       // child changes values
      sprintf(out, "   From child: i=%d, j=%d\n", i, j);
      write(1, out, strlen(out));
   }
   else {                       // parent here
      sleep(3);
      sprintf(out, "From parent: i=%d, j=%d\n", i, j);
      write(1, out, strlen(out));
   }
}
```

force the parent to print after the child terminated

Example 3: 2/2

```
HILBERT:/home/csdept/shene/CS3331/O3-Process(46) fork-3
   From child: i = 1000, j = 2000
From parent: i = 10, j = 20
HILBERT:/home/csdept/shene/CS3331/O3-Process(47)
```

child changes `i` and `j` to 1000 and 2000 from 10 and 20, respectively

parent's `i` and `j` are not affected because parent and child have independent and separate address space

The wait() System Call

- The `wait()` system call blocks the caller until **one of its child processes** exits or a signal is received.

- `wait()` takes a pointer to an integer variable and returns the process ID of the completed process. If no child process is running, `wait()` returns `-1`.

- Some flags that indicate the completion status of the child process are passed back with the integer pointer.

How to Use wait()?

- **Wait for an unspecified child process:**

```
wait(&status);
```

- **Wait for a number, say n, of unspecified child processes:**

```
for (i = 0; i < n; i++)
    wait(&status);
```

- **Wait for a specific child process whose ID is known:**

```
while (pid != wait(&status))
    ;
```

wait() System Call Example

```
void main(void)
{
    pid_t pid, pid_child;
    int   status;

    if ((pid = fork()) == 0)  // child here
        child();
    else {                     // parent here
        parent();
        pid_child = wait(&status);
    }
}
```

The exec() System Calls

- **A newly created process may run a different program rather than that of the parent.**
- **This is done using the `exec` system calls. We will only discuss `execvp()`:**

  ```
  int execvp(char *file, char *argv[]);
  ```

  - ❖ **`file` is a `char` array that contains the name of an executable file**
  - ❖ **`argv[]` is the argument passed to your main program**
  - ❖ **`argv[0]` is a pointer to a string that contains the program name**
  - ❖ **`argv[1]`, `argv[2]`, … are pointers to strings that contain the arguments**

execvp() : An Example 1/2
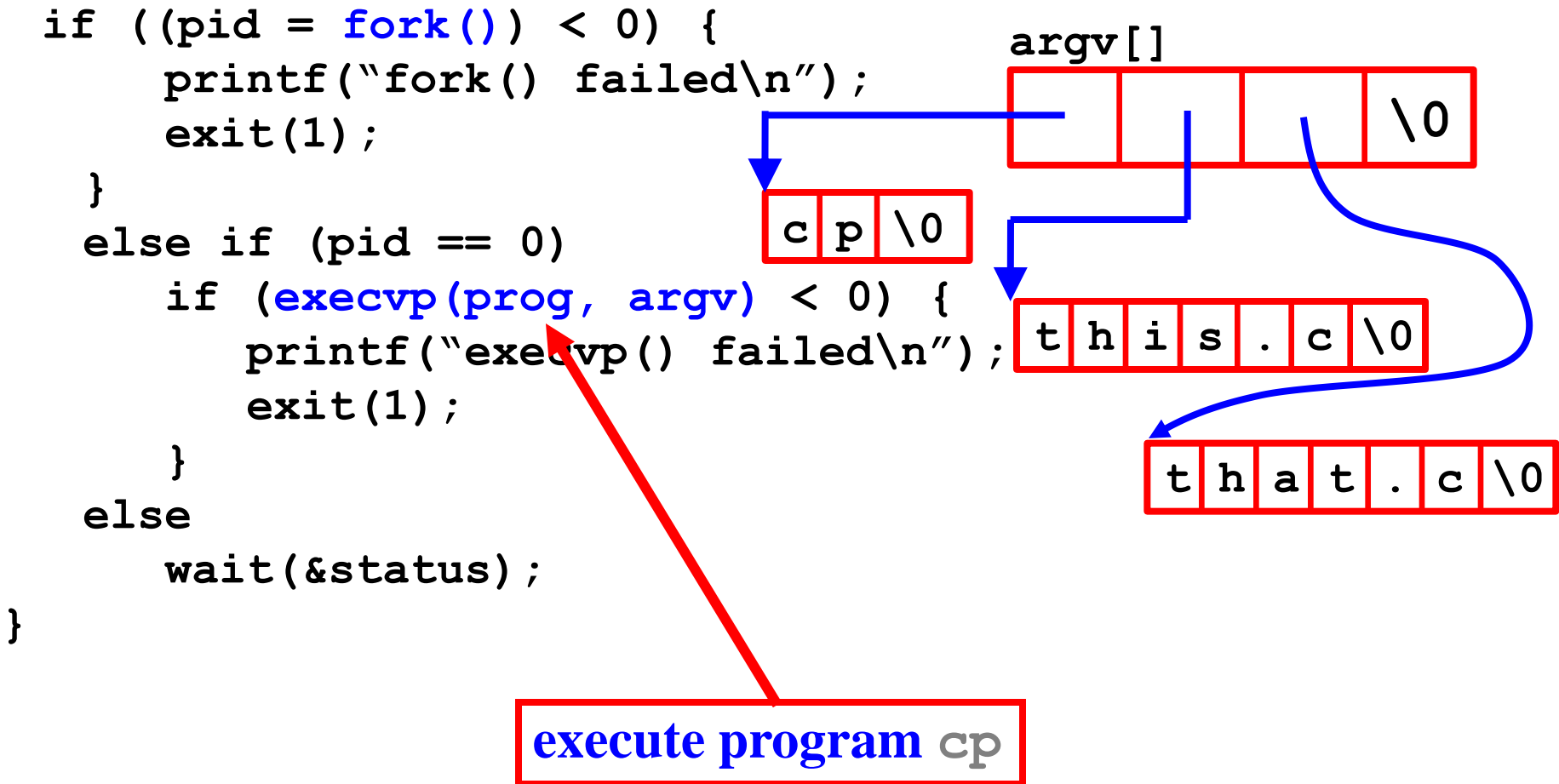
```
#include  <stdio.h>
#include  <unistd.h>

void main(void)
{
  char   prog[] = { "cp" };
  char   in[]   = { "this.c" };
  char   out[]  = { "that.c" };
  char   *argv[4];
  int    status;
  pid_t  pid;

  argv[0] = prog;   argv[1] = in;
  argv[2] = out;    argv[3] = '\0';
          // see next slide
```

argv[]

| | | | \0 |

c p \0

t h i s . c \0

t h a t . c \0

execvp() : An Example 2/2

```
if ((pid = fork()) < 0) {
    printf("fork() failed\n");
    exit(1);
 }
 else if (pid == 0)
    if (execvp(prog, argv) < 0) {
       printf("execvp() failed\n");
       exit(1);
    }
 else
    wait(&status);
}
```

argv[]

c p \0

t h i s . c \0

t h a t . c \0

execute program cp

A Mini-Shell: 1/3

```c
void parse(char *line, char **argv)
{
    while (*line != '\0') {    // not EOLN
        while (*line == ' ' || *line == '\t' || *line == '\n')
            *line++ = '\0';    // replace white spaces with 0
        *argv++ = line;        // save the argument position
        while (*line != '\0' && *line ! =' '
                            && *line!='\t' && *line != '\n')
            line++;            // skip the argument until ...
    }
    *argv = '\0';              // mark the end of argument list
}
```

line[]

| c | p | | t | h | i | s | . | c | | t | h | a | t | . | c | \0 |

line[]

| c | p | \0 | t | h | i | s | . | c | \0 | t | h | a | t | . | c | \0 |

| _ | _ | _ | \0 |

argv[]

A Mini-Shell: 2/3

```
void execute(char **argv)
{
   pid_t pid;
   int status;
   if ((pid = fork()) < 0) { // fork a child process
      printf("*** ERROR: forking child process failed\n");
      exit(1);
   }
   else if (pid == 0) {       // for the child process:
      if (execvp(*argv, argv) < 0) { // execute the command
         printf("*** ERROR: exec failed\n");
         exit(1);
      }
   }
   else {                     // for the parent:
      while (wait(&status) != pid) // wait for completion
         ;
   }
}
```

A Mini-Shell: 3/3

```c
void main(void)
{
    char line[1024];   // the input line
    char *argv[64];    // the command line argument

    while (1) {           // repeat until done ....
        printf("Shell -> "); // display a prompt
        gets(line);       // read in the command line
        printf("\n");
        parse(line, argv);   // parse the line
        if (strcmp(argv[0], "exit") == 0) // is it an "exit"?
            exit(0);      // exit if it is
        execute(argv); // otherwise, execute the command
    }
}
```

Don't forget that `gets()` is risky! Use `fgets()` instead.

What is Shared Memory?

- **The parent and child processes are run in independent and separate address spaces. All processes, parent and children included, do not share anything.**

- **A shared memory segment is a piece of memory that can be allocated and attached to an address space. Processes that have this memory segment attached will have access to it.**

- **But, race conditions can occur!**

Procedure for Using Shared Memory

- **Find a key. Unix uses this key for identifying shared memory segments.**

- **Use `shmget()` to allocate a shared memory.**

- **Use `shmat()` to attach a shared memory to an address space.**

- **Use `shmdt()` to detach a shared memory from an address space.**

- **Use `shmctl()` to deallocate a shared memory.**

Keys: 1/2

- **To use shared memory, include the following:**

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

- **A key is a value of type `key_t`. There are three ways to generate a key:**
  - ❖**Do it yourself**
  - ❖**Use function `ftok()`**
  - ❖**Ask the system to provide a private key.**

Keys: 2/2

- **Do it yourself:**

  ```
  key_t      SomeKey;
  SomeKey = 1234;
  ```

- **Use `ftok()` to generate one for you:**

  ```
  key_t = ftok(char *path, int ID);
  ```

  - ❖ **`path` is a path name (e.g., "./")**
  - ❖ **`ID` is an integer (e.g., 'a')**
  - ❖ **Function `ftok()` returns a key of type `key_t`:**

    ```
    SomeKey = ftok("./", 'x');
    ```

- **Keys are global entities. If other processes know your key, they can access your shared memory.**

- **Ask the system to provide a private key with `IPC_PRIVATE`.**

Ask for a Shared Memory: 1/4

- **Include the following:**

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

- **Use `shmget()` to request a shared memory:**

  ```
  shm_id = shmget(
    key_t  key,    /* identity key    */
    int    size,   /* memory size     */
    int    flag);  /* creation or use */
  ```

- **`shmget()` returns a shared memory ID.**

- **The flag, for our purpose, is either `0666` (rw) or `IPC_CREAT | 0666`. Yes, `IPC_CREAT`.**

- **The following creates a shared memory of size `struct Data` with a private key `IPC_PRIVATE`. This is a creation (`IPC_CREAT`) with read and write permission (`0666`).**

```
struct Data { int a; double b; char x; };
int          ShmID;


ShmID = shmget(
        IPC_PRIVATE,  /* private key */
        sizeof(struct Data), /* size */
        IPC_CREAT | 0666);/* cr & rw */
```

Ask for a Shared Memory: 3/4

- **The following creates a shared memory with a key based on the current directory:**

```
struct Data { int a; double b; char x;};
int      ShmID;
key_t  Key;


Key = ftok("./", 'h');
ShmID = shmget(
          Key,              /* a key */
          sizeof(struct Data),
          IPC_CREAT | 0666);
```

Ask for a Shared Memory: 4/4

- **When asking for a shared memory, the process that creates it uses `IPC_CREAT | 0666` and processes that access a created one use `0666`.**

- **If the return value is negative (Unix convention), the request was unsuccessful, and no shared memory is allocated.**

- **Create a shared memory before its use!**

After the Execution of shmget()

Process 1

Process 2

`shmget(…,IPC_CREAT|0666);`

shared memory

Shared memory is allocated; but, is not part of the address space

Attaching a Shared Memory: 1/3

- **Use `shmat()` to attach an existing shared memory to an address space:**

```
shm_ptr = shmat(
    int  shm_id, /* ID from shmget() */
    char *ptr,   /* use NULL here    */
    int  flag);  /* use 0 here       */
```

- **`shm_id` is the shared memory ID returned by `shmget()`.**

- **Use `NULL` and `0` for the second and third arguments, respectively.**

- **`shmat()` returns a `void` pointer to the memory. If unsuccessful, it returns a negative integer.**

Attaching a Shared Memory: 2/3

```
struct Data { int a; double b; char x;};
int     ShmID;
key_t  Key;
struct Data *p;

Key = ftok("./", 'h');
ShmID = shmget(Key, sizeof(struct Data),
                    IPC_CREAT | 0666);
p = (struct Data *) shmat(ShmID, NULL, 0);
if ((int) p < 0) {
    printf("shmat() failed\n"); exit(1);
}
p->a = 1; p->b = 5.0; p->x = '.';
```
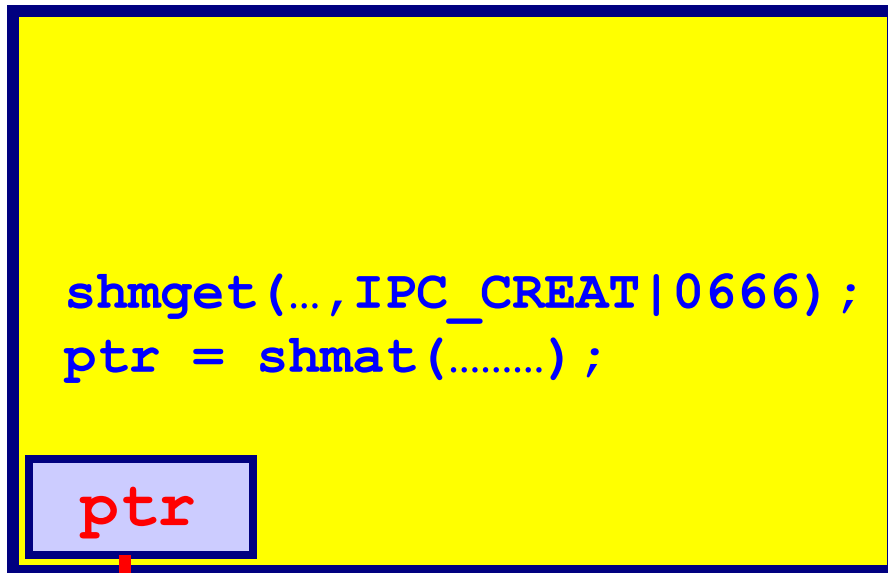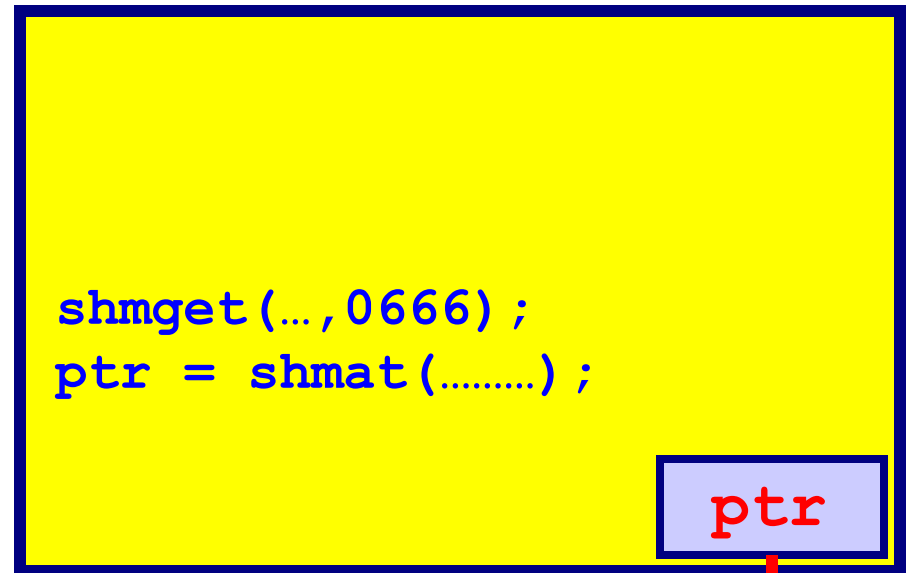
Attaching a Shared Memory: 3/3

Process 1

```
shmget(…,IPC_CREAT|0666);
ptr = shmat(………);
```

ptr

Process 2

```
shmget(…,0666);
ptr = shmat(………);
```

ptr

shared memory

Now processes can access the shared memory

Detaching/Removing Shared Memory

- **To detach a shared memory, use**
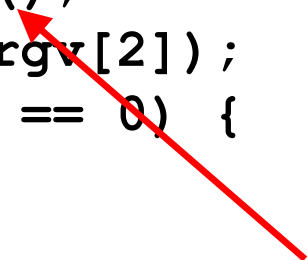
    `shmdt(shm_ptr);`

    `shm_ptr` **is the pointer returned by** `shmat()`.

- **After a shared memory is detached, it is still there. You can attach and use it again.**

- **To remove a shared memory, use**

    `shmctl(shm_ID, IPC_RMID, NULL);`

    `shm_ID` **is the shared memory ID returned by** `shmget()`. **After a shared memory is removed, it no longer exists.**

Communicating with a Child: 1/2

```c
void main(int argc, char *argv[])
{
  int    ShmID, *ShmPTR, status;
  pid_t pid;

  ShmID = shmget(IPC_PRIVATE,4*sizeof(int),IPC_CREAT|0666);
  ShmPTR = (int *) shmat(ShmID, NULL, 0);
  ShmPTR[0] = getpid();       ShmPTR[1] = atoi(argv[1]);
  ShmPTR[2] = atoi(argv[2]);  ShmPTR[3] = atoi(argv[3]);
  if ((pid = fork()) == 0) {
     Child(ShmPTR);
     exit(0);
  }
  wait(&status);
  shmdt((void *) ShmPTR);   shmctl(ShmID, IPC_RMID, NULL);
  exit(0);
}
```

parent's process ID here

Communicating with a Child: 2/2

```
void Child(int  SharedMem[])
{
   printf("%d %d %d %d\n",  SharedMem[0],
       SharedMem[1], SharedMem[2], SharedMem[3]);
}
```

- **Why are** `shmget()` **and** `shmat()` **not needed in the child process?**

Communicating Among Separate Processes: 1/5

- **Define the structure of a shared memory segment as follows:**

```
#define  NOT_READY  (-1)
#define  FILLED     (0)
#define  TAKEN      (1)

struct Memory {
   int  status;
   int  data[4];
};
```

Communicating Among Separate Processes: 2/5

The "Server"

Prepare for a shared memory

```
int main(int argc, char *argv[])
{
    key_t          ShmKEY;
    int            ShmID, i;
    struct Memory  *ShmPTR;

    ShmKEY = ftok("./", 'x');
    ShmID = shmget(ShmKEY, sizeof(struct Memory),
                   IPC_CREAT | 0666);
    ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
```

Communicating Among Separate Processes: 3/5

shared memory not ready

```
ShmPTR->status = NOT_READY;
```

filling in data

```
ShmPTR->data[0] = getpid();
for (i = 1; i < 4; i++)
    ShmPTR->data[i] = atoi(argv[i]);
```

```
ShmPTR->status = FILLED;
while (ShmPTR->status != TAKEN)
    sleep(1);   /* sleep for 1 second */
printf("My buddy is %ld\n", ShmPTR->data[0]);
shmdt((void *) ShmPTR);
shmctl(ShmID, IPC_RMID, NULL);
exit(0);
}
```

wait until the data is taken

detach and remove shared memory

54

Communicating Among Separate Processes: 4/5

```
int main(void)
{                                      The "Client"
    key_t           ShmKEY;
    int             ShmID;          prepare for shared memory
    struct Memory   *ShmPTR;

    ShmKEY=ftok("./", 'x');
    ShmID = shmget(ShmKEY, sizeof(struct Memory), 0666);
    ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
    while (ShmPTR->status != FILLED)
        ;
    printf("%d %d %d %d\n", ShmPTR->data[0],
        ShmPTR->data[1], ShmPTR->data[2], ShmPTR->data[3]);
    ShmPTR->data[0] = getpid();
    ShmPTR->status = TAKEN;
    shmdt((void *) ShmPTR);
    exit(0);
}
```

Communicating Among Separate Processes: 5/5

- **The "server" must run first to prepare a shared memory.**

- **Run the server in one window, and run the client in another a little later.**

- **Or, run the server as a background process. Then, run the client in the foreground later:**

  ```
  server 1 3 5 &
  client
  ```
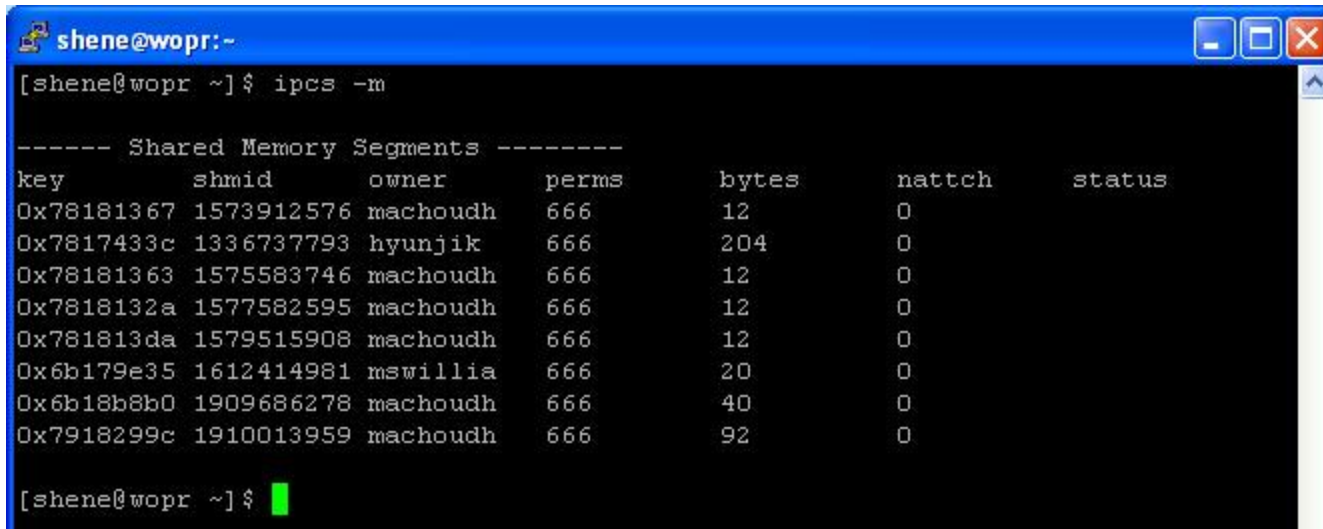
- **This version uses busy waiting.**

- **One may use Unix semaphores for mutual exclusion.**

Important Notes: 1/3

- **If you do not remove your shared memory segments (e.g., program crashes before the execution of `shmctl()`), they will be in the system forever until the system is shut down. This will degrade the system performance.**

- **Use the `ipcs` command to check if you have shared memory segments left in the system.**

- **Use the `ipcrm` command to remove your shared memory segments.**

Important Notes: 2/3

- **To see existing shared memory segments in the system, use `ipcs -m`, where `m` means shared memory.**

- **The following is a snapshot on `wopr`:**

```
shene@wopr:~
[shene@wopr ~]$ ipcs -m

------ Shared Memory Segments ---------
key        shmid       owner     perms   bytes    nattch    status
0x78181367 1573912576  machoudh  666     12       0
0x7817433c 1336737793  hyunjik   666     204      0
0x78181363 1575583746  machoudh  666     12       0
0x7818132a 1577582595  machoudh  666     12       0
0x781813da 1579515908  machoudh  666     12       0
0x6b179e35 1612414981  mswillia  666     20       0
0x6b18b8b0 1909686278  machoudh  666     40       0
0x7918299c 1910013959  machoudh  666     92       0

[shene@wopr ~]$
```

Important Notes: 3/3

- **To remove a shared memory, use the `ipcrm` command as follows:**
  - ❖ `ipcrm -M` **shm-key**
  - ❖ `ipcrm -m` **shm-ID**
- **You have to be the owner (or super user) to remove a shared memory.**

The End