Introduction to Process

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Introduction to Process

Contents

Basic Concepts

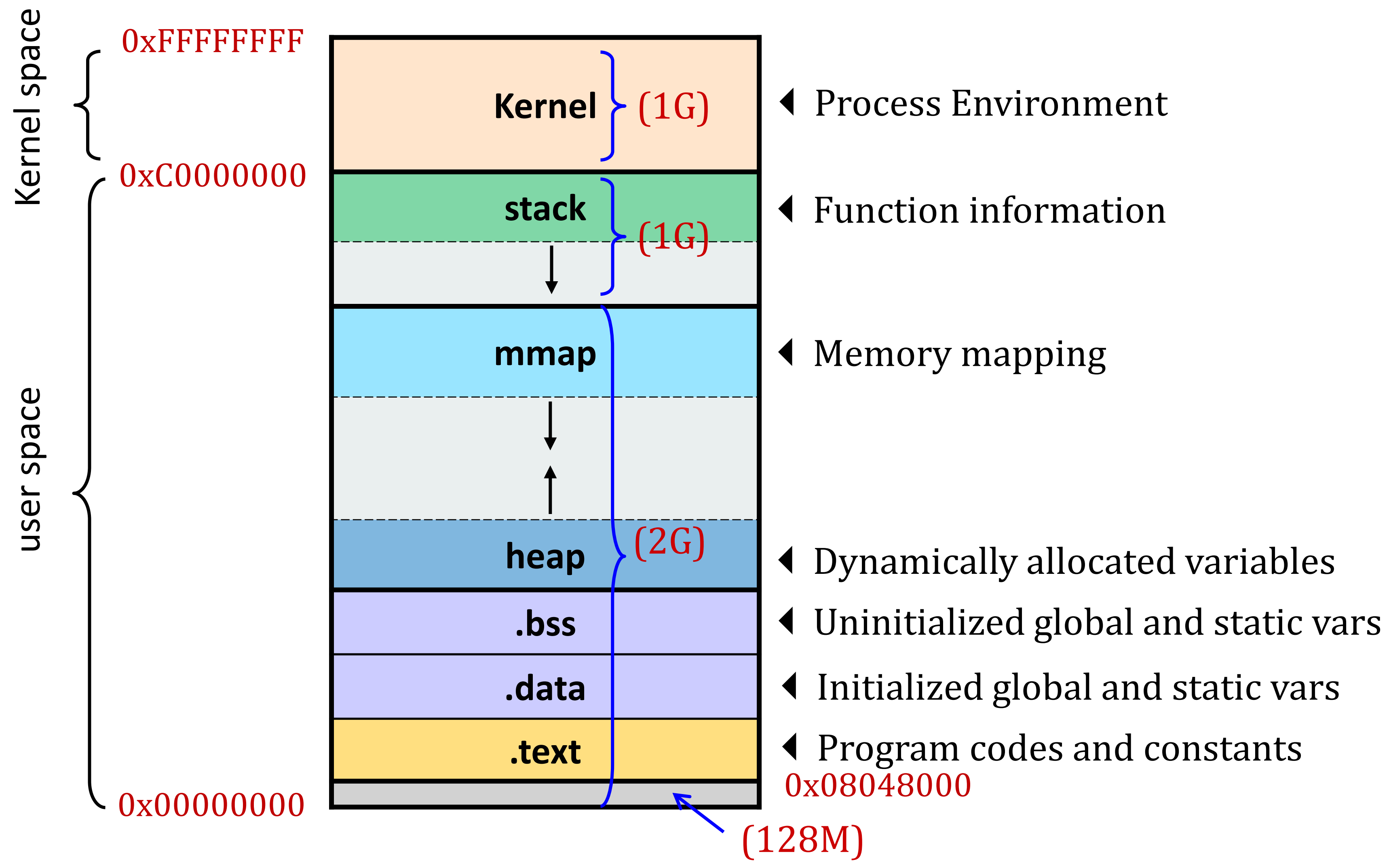
- Process is a program in execution; forms the basis of all computation. Process execution must progress in sequential fashion.
  - A program is a passive entity containing a list of instructions stored on disk as an executable file.
  - A process is an active entity with some specifications of the corresponding program and a set of associated resources.
  - A program becomes a process when the executable object of this program is loaded into memory, given to the process scheduler.
  - A process is an instance of a running program; it can be assigned to, and executed on, a processor.
- Execution of program can be started via CLI entry of its name, GUI mouse clicks, etc.
- Related terms for Process
  - Job, Step, Load Module, Task, Thread.

Basic Concepts

- A process includes some segments/sections:
  - Text
    - the executable (program) code
  - Data & Heap
    - Data
      - global variables
    - Heap
      - memory dynamically allocated during run time
  - Stack
    - temporary data storage
      - procedure/function parameters, return addresses, local variables
- Current activity of a program, or a process, includes its context.
  - program counter (PC), processor registers, etc.
- One program can be corresponding to several processes.
  - multiple users executing the same sequential program
  - concurrent program running several processes

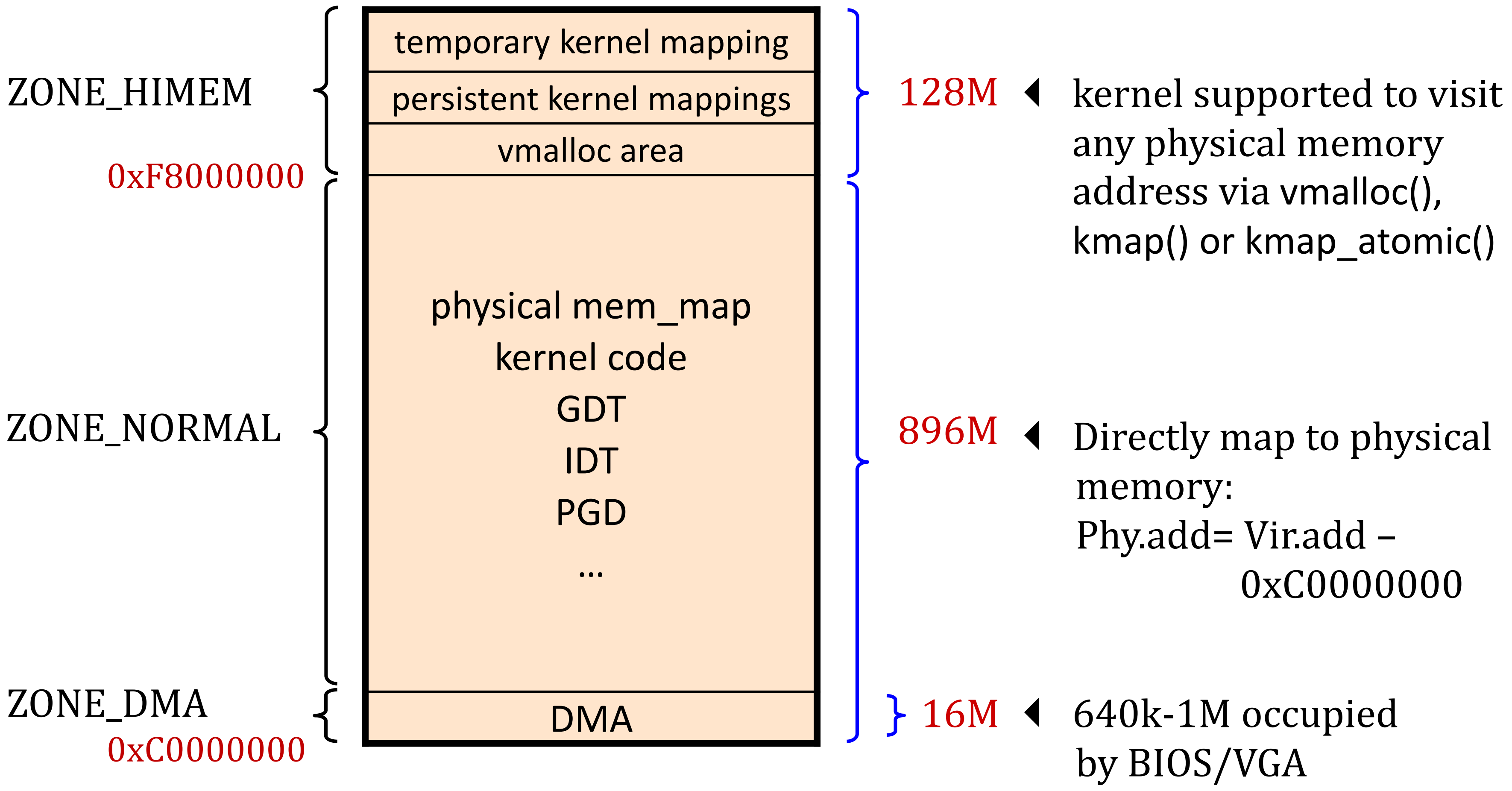
Basic Concepts

- Process Virtual Memory - Typical layout on Linux/IA-32.



Basic Concepts

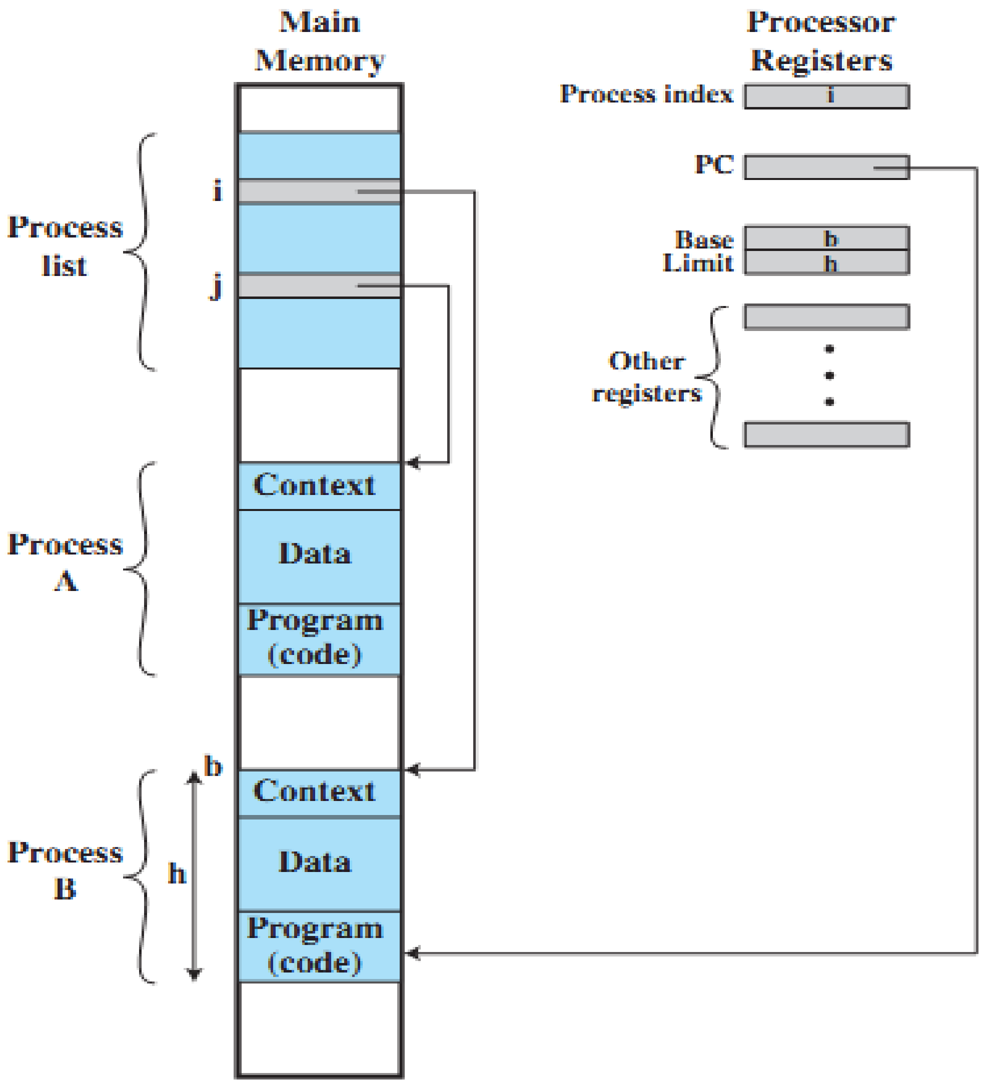
- Process Virtual Memory - Typical layout on Linux/IA-32.


ZONE_HIMEM
0xF8000000
ZONE_NORMAL
ZONE_DMA
0xC0000000
temporary kernel mapping
persistent kernel mappings
vmalloc area
physical mem_map
kernel code
GDT
IDT
PGD
...
DMA
128M
896M
16M
kernel supported to visit any physical memory address via vmalloc(), kmap() or kmap_atomic()
Directly map to physical memory:
Phy.add= Vir.add – 0xC0000000
640k-1M occupied by BIOS/VGA

Kernel space layout

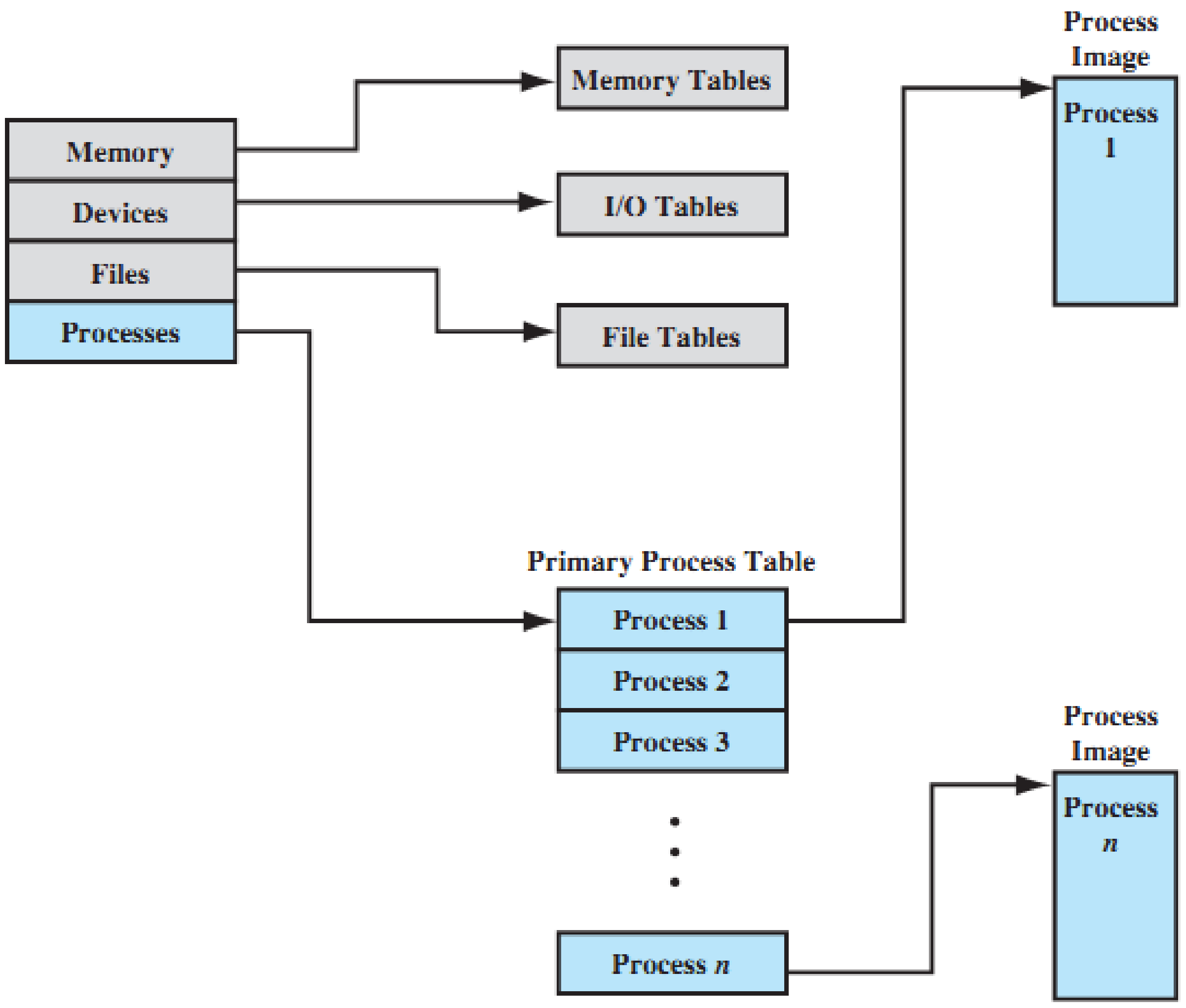
Basic Concepts

- Typical Process Table Implementation.



Basic Concepts

- General Structure of OS Control Tables.



Basic Concepts

- **Basic Concepts**
  - Attributes of Process
    - Process ID
    - Parent process ID
    - User ID
    - Process state
    - Process priority
    - Program counter
    - CPU registers
    - Memory management information
    - I/O status information
    - Access control
    - Accounting information

Process Table and PCB

Process Table

- *Process table* is a kernel data structure containing fields that must always be available to the kernel.
  - state field (that identifies the state of the process)
  - fields that allow kernel to locate the process in memory
  - UIDs for determining various process privileges
  - PIDs to specify relationships b/w processes (e.g. fork)
  - event descriptor (when the process in sleep state)
  - scheduling parameters to determine the order in which process moves to the states "kernel running" and "user running"
  - signal field for signals send to the process but not yet handled
  - timers that give process execution time in kernel mode and user mode
  - field that gives process size (so that kernel knows how much space to allocate for the process).

Process Table and PCB

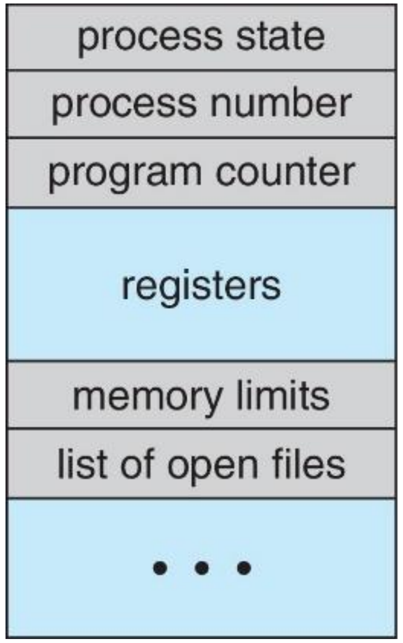
Process Table

- Fields of a Typical Process Table.

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program Counter	Pointer to data segment info	Working directory
Program Status Word	Pointer to stack segment info	File descriptors
Stack Pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Process Table and PCB

- **Process Control Block (PCB)**
  - Each process is represented in OS by a *process control block* (PCB)
    - PCB also called a *task control block*, IBM name for information associated with a specific process.
      - It saves context of the Process.
  - PCB is the data (Process Attributes) needed by OS to control process:
    - Process location information
    - Process identification information
    - Processor state information
    - Process control information.



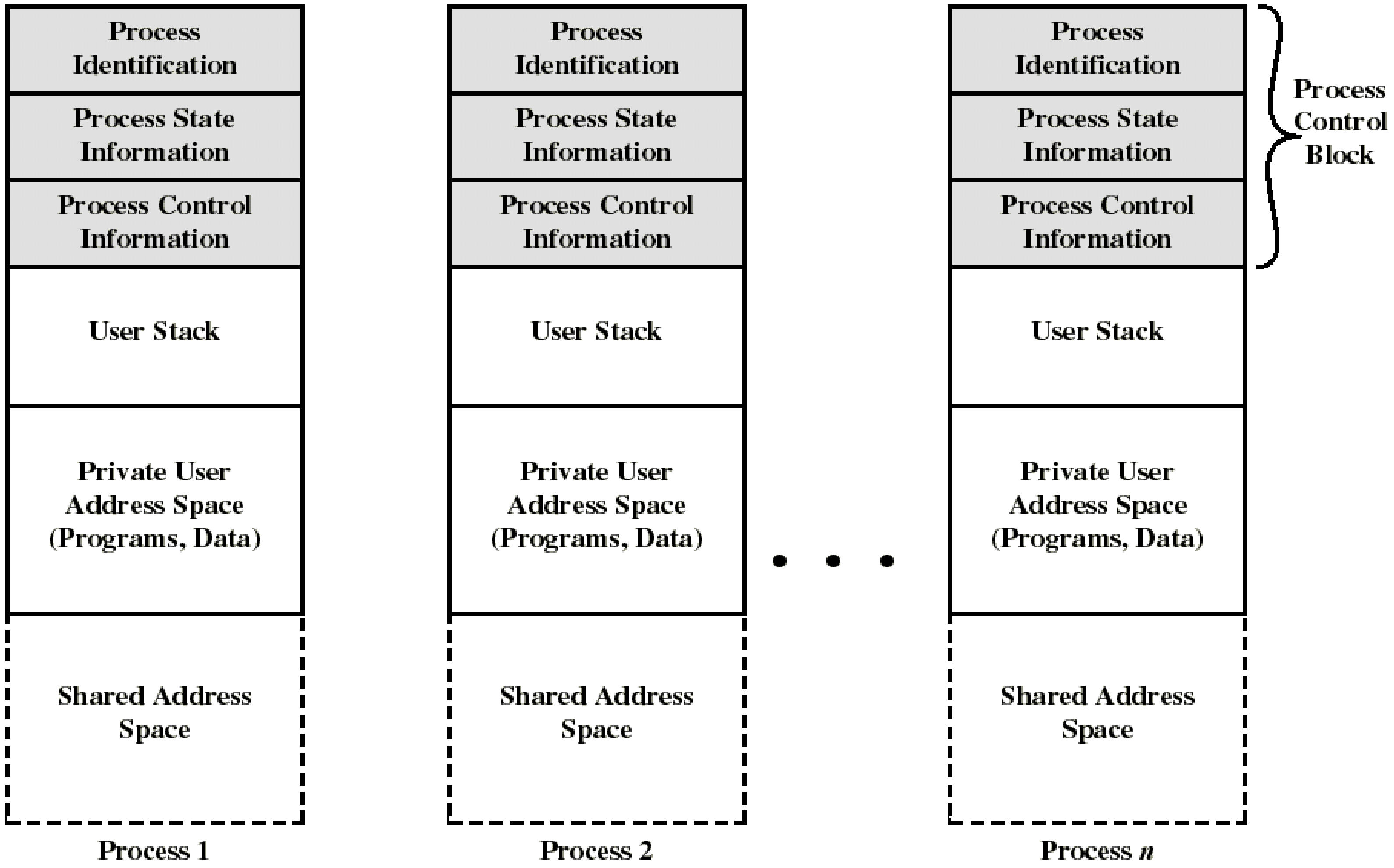
Process control block (PCB)

Process Table and PCB

- **Process Control Block (PCB)**
  - Process Location Information
    - Process image
      - Each process has an image in memory.
      - It may not occupy a contiguous range of addresses.
        - depends on memory management scheme used
      - Both a private and shared memory address space can be used.
    - Each process image is pointed to by an entry in the process table.
    - For the OS to manage the process, at least part of its image must be brought into main memory.

Process Table and PCB

- **Process Control Block (PCB)**
  - Process Location Information.



Process Images in Memory

Process Table and PCB

- **Process Control Block (PCB)**
  - Process Identification Information
    - A few numeric identifiers may be used:
      - Unique process identifier (PID)
        - indexes (directly or indirectly) into the process table
      - User identifier (UID)
        - the user who is responsible for the job
      - Identifier of the process that created this process (PPID, Parent process ID)
    - Maybe symbolic names that are related to numeric identifiers

Process Table and PCB

- **Process Control Block (PCB)**
  - Processor State Information
    - contents of processor registers
      - User-visible registers
      - Control and status registers
      - Stack pointers.
    - Program Status Word (PSW)
      - contains status information
      - E.g.
        - the EFLAGS register on Pentium machines

Process Table and PCB

- **Process Control Block (PCB)**
  - Process Control Information
    - Scheduling and state information
      - *Process state* (e.g., running, ready, blocked...)
      - Priority of the process
      - Event for which the process is waiting (if blocked)
    - Data structuring information
      - may hold pointers to other PCBs for process queues, parent-child relationships and other structures
    - InterProcess Communication (IPC)
      - may hold flags and signals for IPC
    - Resource ownership and utilization
      - resource in use: open files, I/O devices...
      - history of usage (of CPU time, I/O...)
    - Process privileges (Access Control)
      - access to certain memory locations, to resources, etc.
    - Memory management
      - pointers to segment/page tables assigned to this process

Process Table and PCB

- **Process Control Block (PCB)**
  - Process Control Information (cont.)
    - Program counter
      - indicates the address of the next instruction to be executed for this process
    - Accounting information
      - includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on
    - I/O status information
      - includes the list of I/O devices allocated to the process, a list of open files, and so on

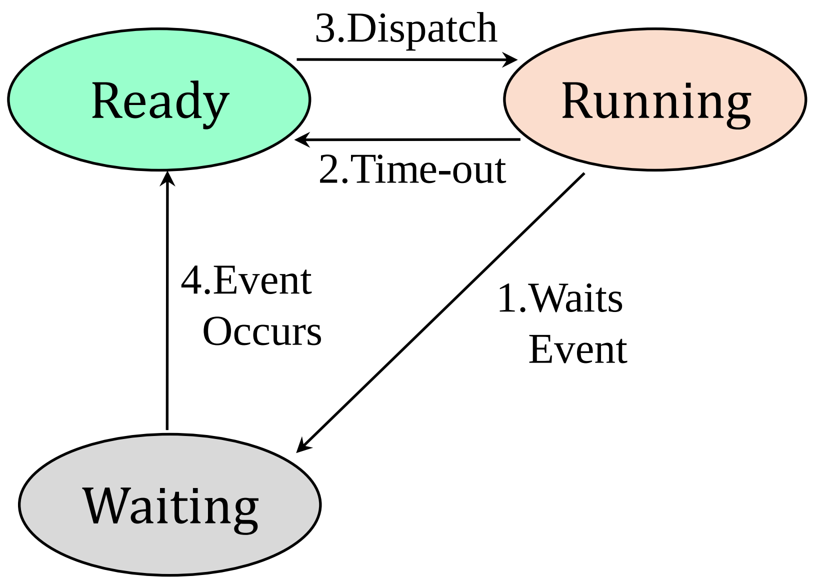
Process States and Transitions

- Three-state Process Model
  - Running State
    - the process that gets executed; its instructions are being executed
  - Ready State
    - any process that is ready to be executed; the process is waiting to be assigned to a processor
  - Waiting/Blocked State
    - any process that cannot execute until its I/O completes or some other event occurs

Process States and Transitions

Process States and Transitions

- Three-state Process Model.



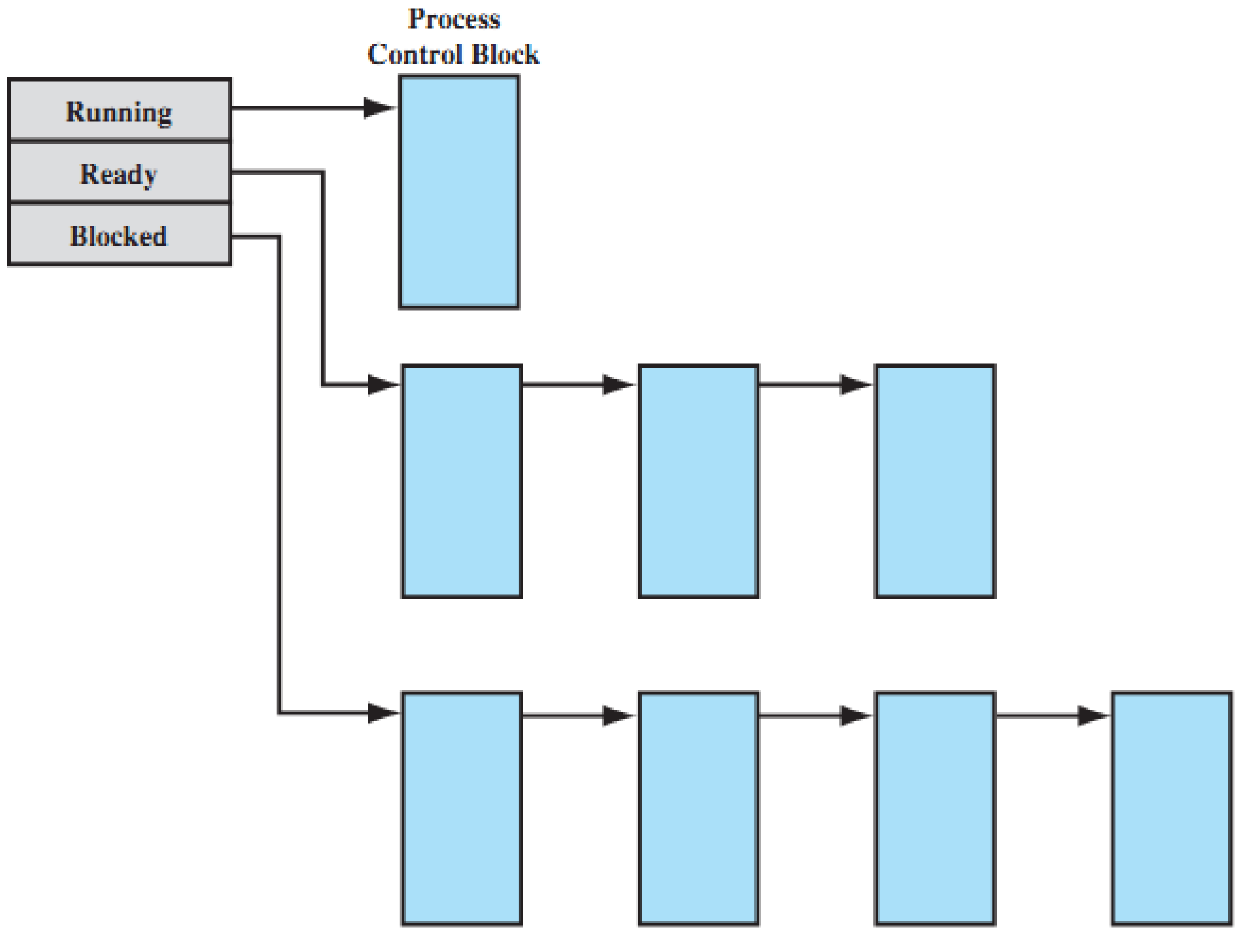
1. Process blocks for input.
2. Scheduler picks another process.
3. Scheduler picks this process.
4. Input becomes available.

Process States and Transitions

- Three-state Process Model
  - Process State Transitions
    - Ready $\rightarrow$ Running
      - When it is time, the dispatcher selects a new process to run.
    - Running $\rightarrow$ Ready
      - The running process has expired his time slot.
      - The running process gets interrupted because a higher priority process is in the ready state.
    - Running $\rightarrow$ Waiting
      - When a process requests something for which it must wait:
        - a service that the OS is not ready to perform
        - an access to a resource not yet available
        - Initiating I/O and waiting for the result
        - waiting for a process to provide input
    - Waiting $\rightarrow$ Ready
      - When the event for which the process was waiting occurs

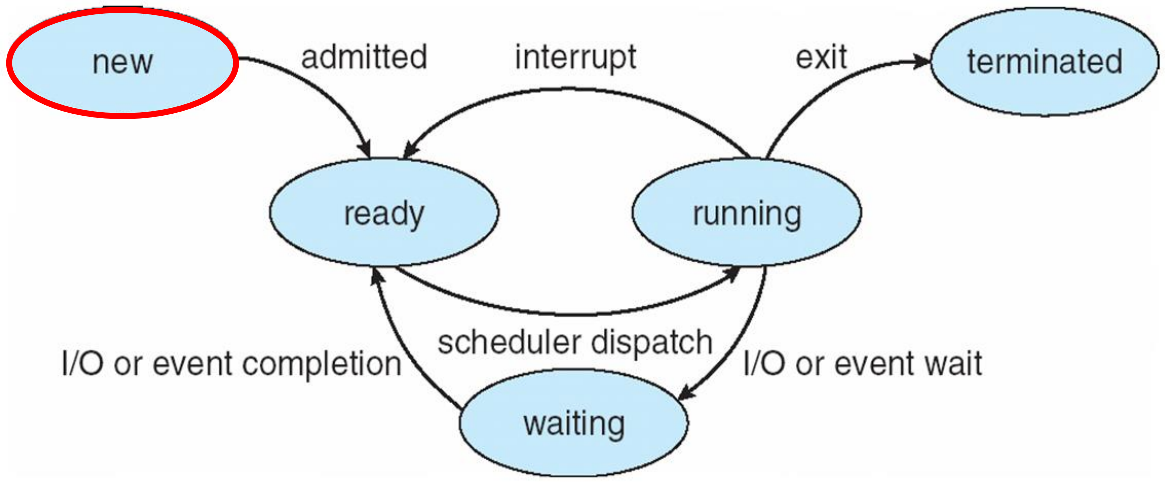
Process States and Transitions

- Three-state Process Model
  - Process List Structures.



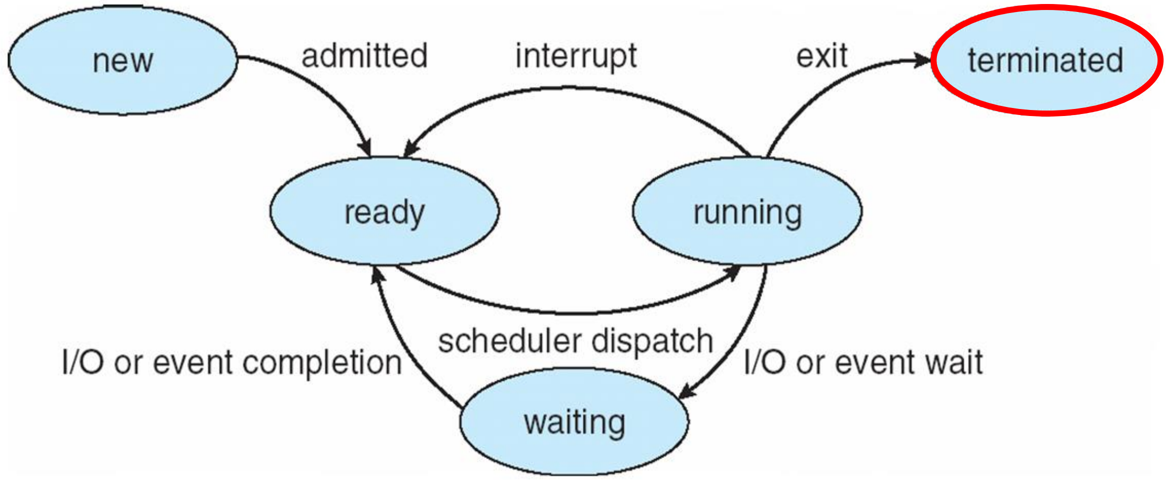
Process States and Transitions

- Five-state Process Model
  - New state
    - OS has performed the necessary actions to create the process:
      - has created a process identifier
      - has created tables needed to manage the process
    - but has not yet committed to execute the process (not yet admitted):
      - because resources are limited



Process States and Transitions

- Five-state Process Model
  - Terminated state
    - Process termination moves the process to terminate state.
      - It is no longer eligible for execution.
    - Tables and other information are temporarily preserved for auxiliary program.
      - E.g., accounting program that cumulates resource usage for billing the users
    - The process (and its tables) gets deleted when the data is no more needed.



Operations on Processes

Process Creation

- When to Create a Process
  - System initialization
  - Submission of a batch job
  - User logs on
  - Created by OS to provide a service to a user
    - e.g., printing a file
  - A user request to create a new process
  - Spawned (繁衍) by an existing process
    - A program can dictate (to require or determine necessarily) the creation of a number of processes.
    - The creating process is the *parent* process and the new processes created are called the *children* of that process.

Operations on Processes

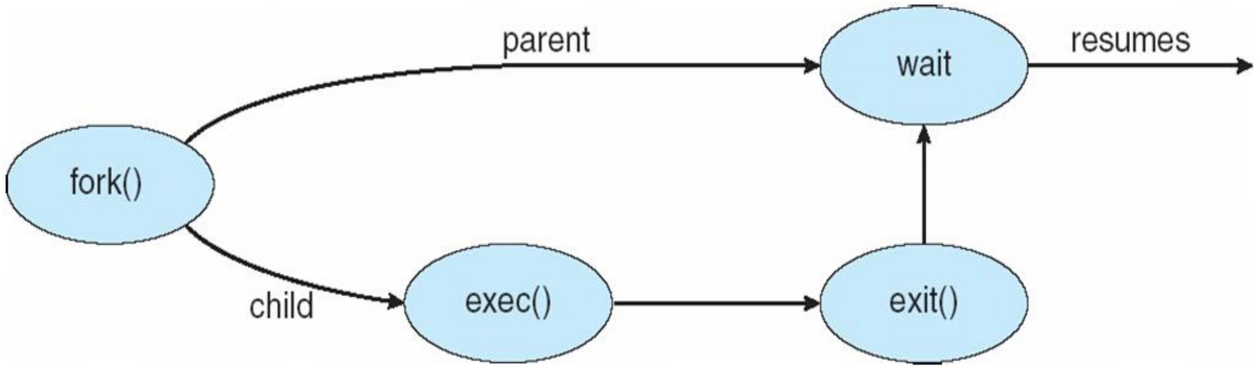
- **Process Creation**
  - Details in Process Creating
    - Parent process create children processes, which, in turn create other processes, forming a *tree* of processes.
    - Possible resource sharing:
      - Parent process and children processes share all resources.
      - Children processes share subset of parent’s resources.
      - Parent process and child process share no resources.
    - Possible execution:
      - Parent process and children processes execute concurrently.
      - Parent process waits until children processes terminate.

Operations on Processes

- **Process Creation**
  - Details in Process Creating (cont.)
    - Assign a unique process identifier (PID).
      - typically an integer number
    - Allocate space for the process image.
    - Initialize Process Control Block (PCB).
      - many default values
        - E.g., state is New, no I/O devices or files, ...
    - Set up appropriate linkages.
      - E.g., add new process to linked list used for the scheduling queue
    - Address space
      - child is a duplicate of parent, or
      - child has a program loaded into it.

Operations on Processes

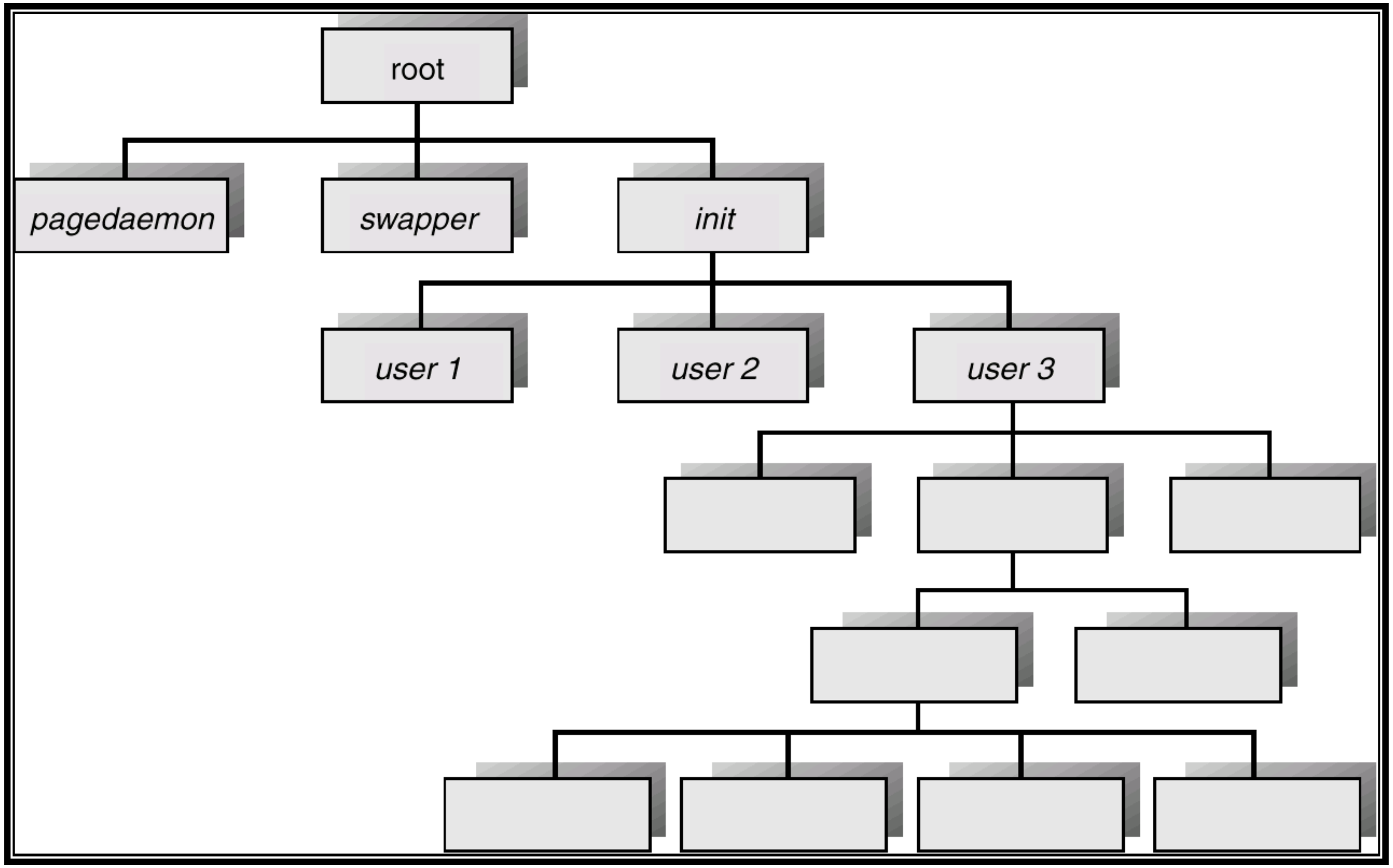
- **Process Creation**
  - Details in Process Creating (cont.)
    - UNIX examples
      - fork() system call creates new process.
      - exec() system call used after a fork() to replace the memory space of the process with a new program.



Operations on Processes

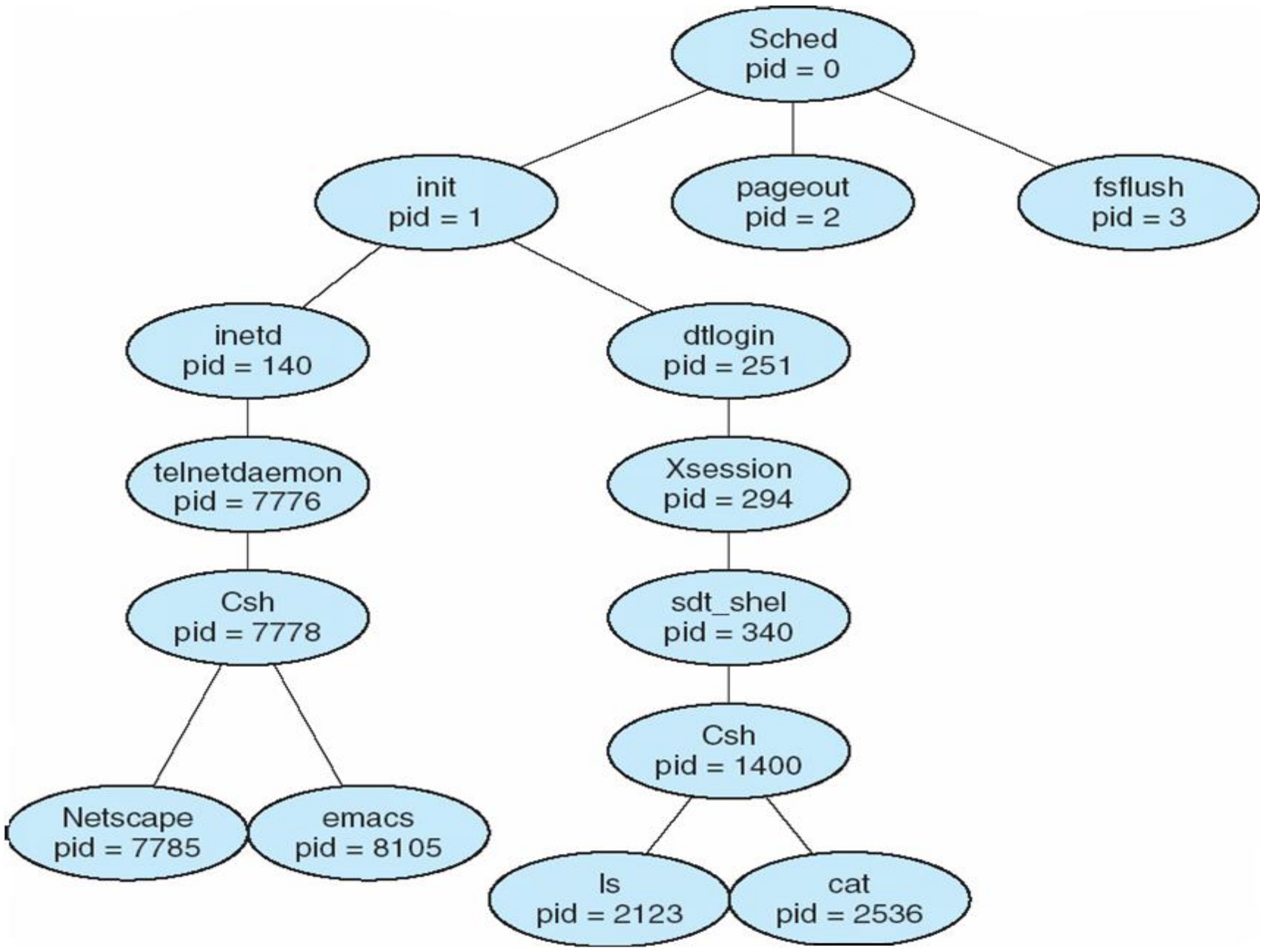
Process Creation

- A Tree of Processes on UNIX.



Operations on Processes

- **Process Creation**
  - A Tree of Processes on Typical Solaris.



Operations on Processes

Process Creation

- Algorithm 6-1: fork-demo.c (forking a separate process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;

    childpid = fork(); /* child duplicates parent’s address space */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    }
    else /* fork() returns 2 values: 0 for child pro and childpid for parent pro */
        if (childpid == 0) { /* This is child pro */
            count++;
            printf(“Child pro pid = %d, count = %d (addr = %p)\n", getpid(), count,
&count);
        }
        else { /* This is parent pro */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            sleep(5);
            wait(0); /* waiting for all children terminated */
        }
    printf("Testing point by %d\n", getpid()); /* child executed this statement and
became defunct before parent wait()*/
    return EXIT_SUCCESS;
}
```

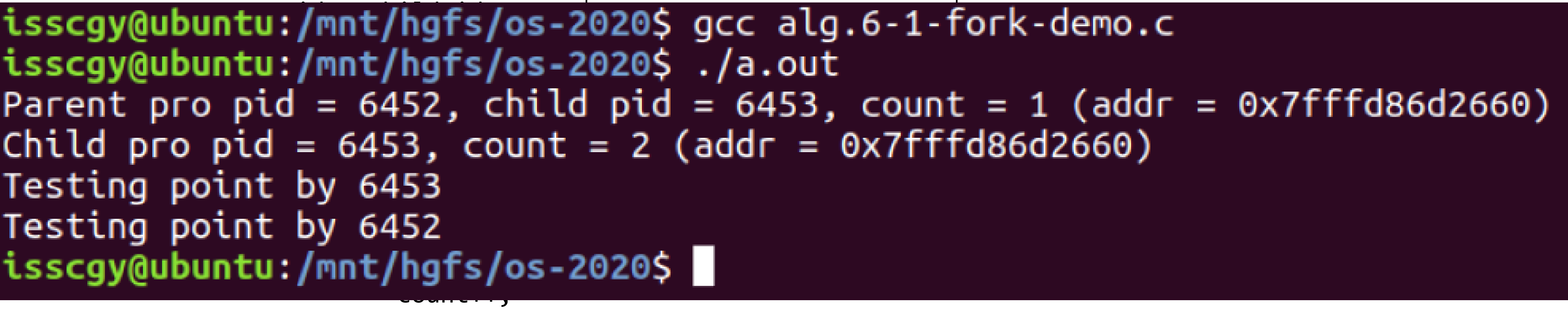
Operations on Processes

Process Creation

- Algorithm 6-1: fork-demo.c (forking a separate process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-1-fork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 6452, child pid = 6453, count = 1 (addr = 0x7fffd86d2660)
Child pro pid = 6453, count = 2 (addr = 0x7fffd86d2660)
Testing point by 6453
Testing point by 6452
isscgy@ubuntu:/mnt/hgfs/os-2020$ 
```

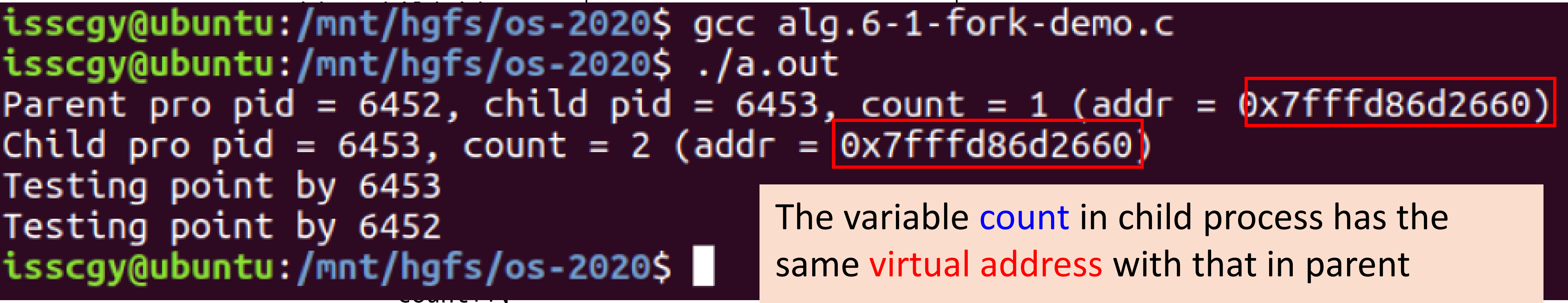
```
            printf(“Child pro pid = %d, count = %d (addr = %p)\n", getpid(), count,
&count);
        }
        else { /* This is parent pro */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            sleep(5);
            wait(0); /* waiting for all children terminated */
        }
    printf("Testing point by %d\n", getpid()); /* child executed this statement and
became defunct before parent wait()*/
    return EXIT_SUCCESS;
}
```

Process Creation

- Algorithm 6-1: fork-demo.c (forking a separate process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-1-fork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 6452, child pid = 6453, count = 1 (addr = 0x7fffd86d2660)
Child pro pid = 6453, count = 2 (addr = 0x7fffd86d2660)
Testing point by 6453
Testing point by 6452
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

The variable count in child process has the same virtual address with that in parent process.

```
            printf("Child pro pid = %
&count);
        }
        else { /* This is parent pro */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n", 
getpid(), childpid, count, &count);
            sleep(5);
            wait(0); /* waiting for all children terminated */
        }
    printf("Testing point by %d\n", getpid()); /* child executed this statement and 
became defunct before parent wait()*/
    return EXIT_SUCCESS;
}
```

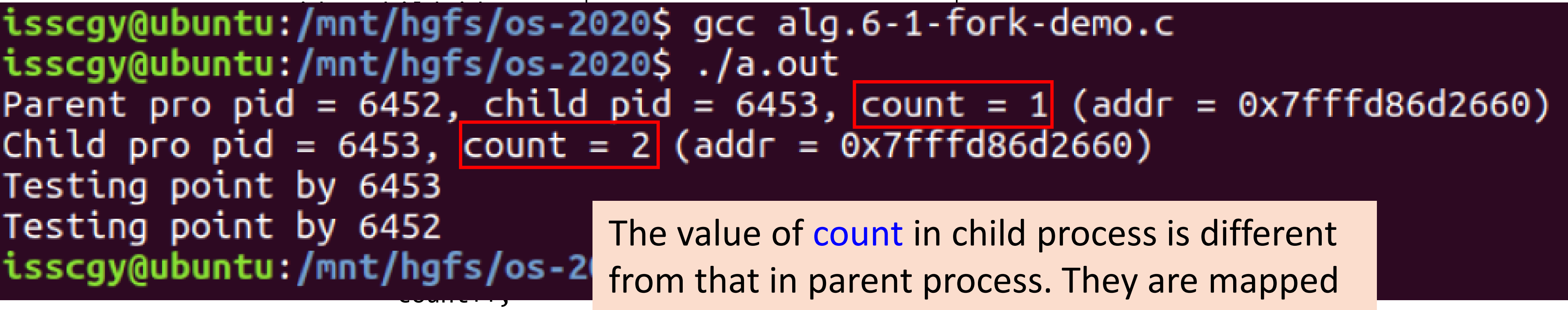
Operations on Processes

Process Creation

- Algorithm 6-1: fork-demo.c (forking a separate process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-1-fork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 6452, child pid = 6453, count = 1 (addr = 0x7fffd86d2660)
Child pro pid = 6453, count = 2 (addr = 0x7fffd86d2660)
Testing point by 6453
Testing point by 6452
isscgy@ubuntu:/mnt/hgfs/os-2
```

The value of count in child process is different from that in parent process. They are mapped to different physical addresses in different process images.

```
            count++;
            printf(“Child                                                nt,
&count);
        }
        else { /* This is parent pro */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            sleep(5);
            wait(0); /* waiting for all children terminated */
        }
    printf("Testing point by %d\n", getpid()); /* child executed this statement and
became defunct before parent wait()*/
    return EXIT_SUCCESS;
}
```

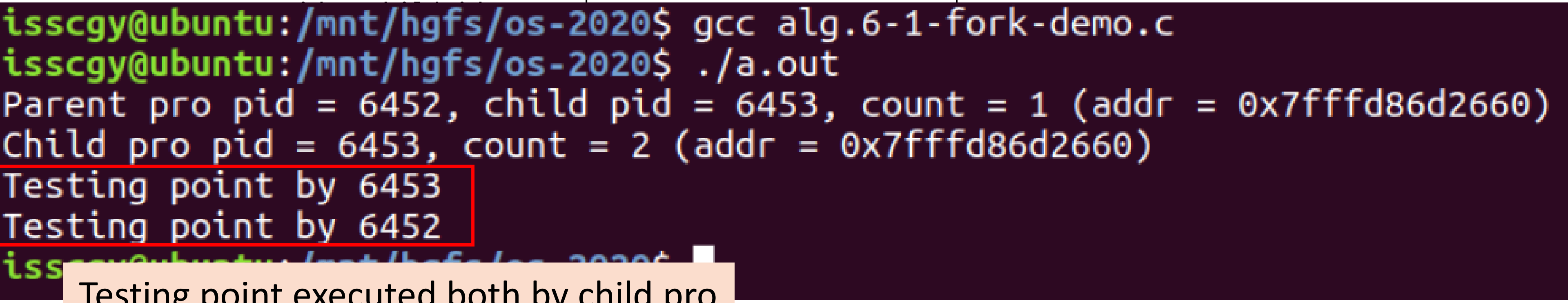
Operations on Processes

Process Creation

- Algorithm 6-1: fork-demo.c (forking a separate process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-1-fork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 6452, child pid = 6453, count = 1 (addr = 0x7fffd86d2660)
Child pro pid = 6453, count = 2 (addr = 0x7fffd86d2660)
Testing point by 6453
Testing point by 6452
```

Testing point executed both by child pro and parent pro

```
%d, count = %d (addr = %p)\n", getpid(), count,
        }
        else { /* This is parent pro */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            sleep(5);
            wait(0); /* waiting for all children terminated */
        }
    printf("Testing point by %d\n", getpid()); /* child executed this statement and
became defunct before parent wait()*/
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **Process Creation**
  - Algorithm 6-2: vfork-demo.c (vforking a sharing-space process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

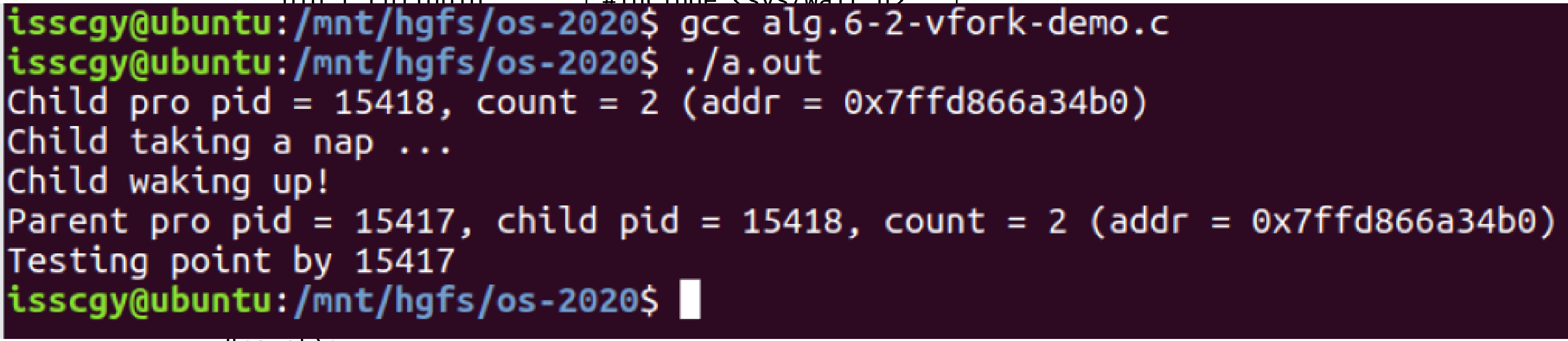
```
int main(void)
{
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* child shares parent’s address space */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    }
    else /* vfork() returns 2 values: 0 for child pro and childpid for parent pro */
        if (childpid == 0) { /* This is child pro, parent hung up until child exit */
            count++;
            printf(“Child pro pid = %d, count = %d (addr = %p)\n", getpid(), count, 
&count);
            printf(“Child taking a nap ...\n");
            sleep(10); printf(“Child waking up!\n");
            _exit(0); /* or exec(0); "return" will cause stack smashing */
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n", 
getpid(), childpid, count, &count);
            wait(0); /* not waitting this vforked child terminated */
        }
    printf("Testing point by %d\n", getpid()); /* executed by parent pro only */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **Process Creation**
  - Algorithm 6-2: vfork-demo.c (vforking a sharing-space process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Child pro pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Child taking a nap ...
Child waking up!
Parent pro pid = 15417, child pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Testing point by 15417
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

```
&count);
            printf("Child taking a nap ...\n");
            sleep(10); printf("Child waking up!\n");
            _exit(0); /* or exec(0); "return" will cause stack smashing */
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            wait(0); /* not waitting this vforked child terminated */
        }
    printf("Testing point by %d\n", getpid()); /* executed by parent pro only */
    return EXIT_SUCCESS;
}
```

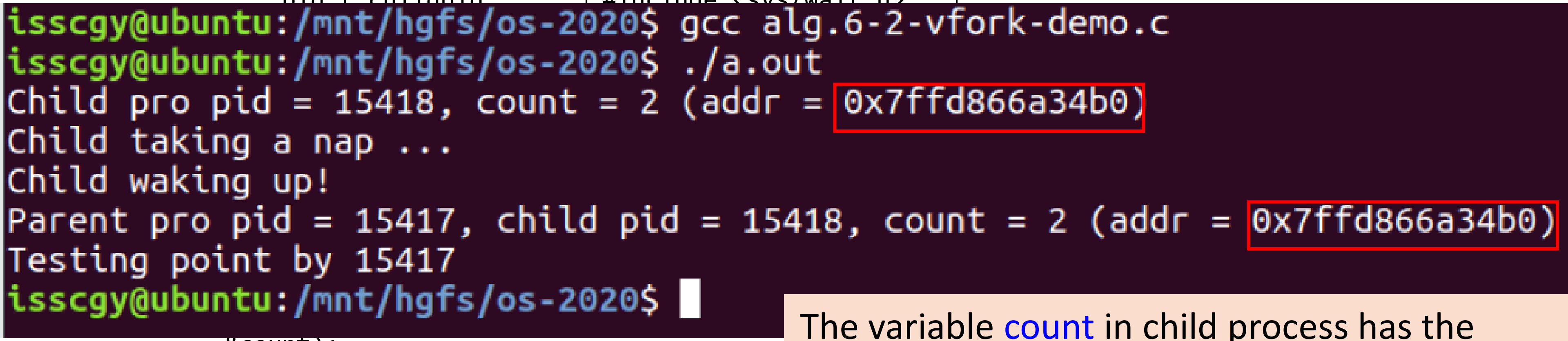
Operations on Processes

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking a sharing-space process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Child pro pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Child taking a nap ...
Child waking up!
Parent pro pid = 15417, child pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Testing point by 15417
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

The variable count in child process has the same virtual address with that in parent process.

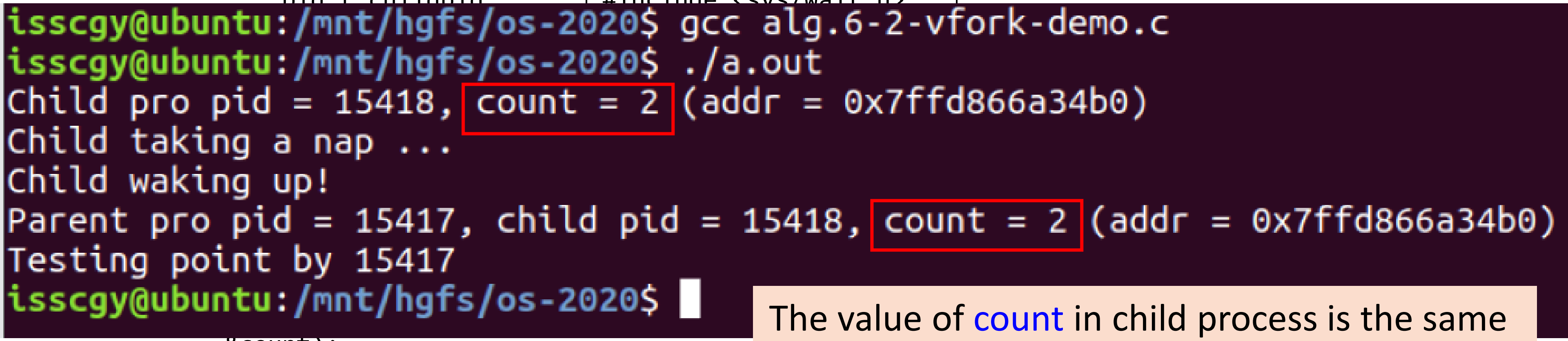
```
&count);
            printf(“Child taking a nap
            sleep(10); printf(“Child wa
            _exit(0); /* or exec(0); "r
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            wait(0); /* not waitting this vforked child terminated */
        }
    printf("Testing point by %d\n", getpid()); /* executed by parent pro only */
    return EXIT_SUCCESS;
}
```

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking a sharing-space process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Child pro pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Child taking a nap ...
Child waking up!
Parent pro pid = 15417, child pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Testing point by 15417
isscgy@ubuntu:/mnt/hgfs/os-2020$ 
```

The value of count in child process is the same as that in parent process. They are mapped to the same physical address in the same process images.

```
&count);
            printf(“Child taking a na
            sleep(10); printf(“Child
            _exit(0); /* or exec(0);
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            wait(0); /* not waitting this vforked child terminated */
        }
    printf("Testing point by %d\n", getpid()); /* executed by parent pro only */
    return EXIT_SUCCESS;
}
```

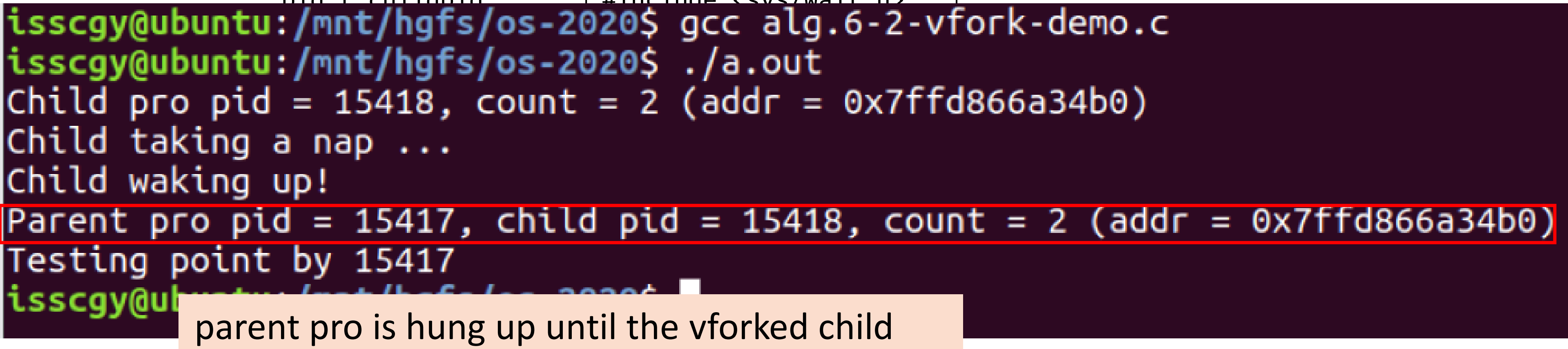
Operations on Processes

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking a sharing-space process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Child pro pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Child taking a nap ...
Child waking up!
Parent pro pid = 15417, child pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Testing point by 15417
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

parent pro is hung up until the vforked child terminated.

```
            sleep(10); printf("Child waking up!\n");
            _exit(0); /* or exec(0); "return" will cause stack smashing */
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n", 
getpid(), childpid, count, &count);
            wait(0); /* not waitting this vforked child terminated */
        }
    printf("Testing point by %d\n", getpid()); /* executed by parent pro only */
    return EXIT_SUCCESS;
}
```

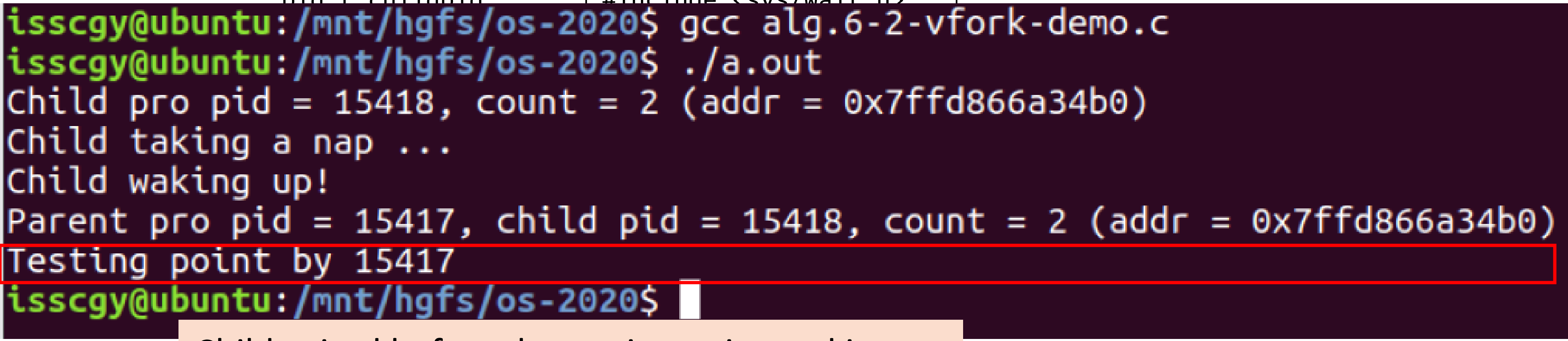
Operations on Processes

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking a sharing-space process).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;
```



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Child pro pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Child taking a nap ...
Child waking up!
Parent pro pid = 15417, child pid = 15418, count = 2 (addr = 0x7ffd866a34b0)
Testing point by 15417
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

Child exited before the testing point and it was executed by parent only

```
                                                                  ;
            _exit(0); /* or exec(0); "return" will cause stack smashing */
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            wait(0); /* not waitting this vforked child terminated */
        }
    printf("Testing point by %d\n", getpid()); /* executed by parent pro only */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **Process Termination**
  - A Process terminates when one of the following events happened
    - Batch job issues Halt instruction.
    - User logs off.
    - Process executes a service request to terminate.
    - Parent kills child process.
    - Error and fault conditions.

Operations on Processes

Process Termination

- Reasons for process termination
  - Normal/Error/Fatal exit
  - Time limit exceeded
  - Time overrun
    - process waited longer than a specified maximum for an event
  - Memory unavailable
  - Memory bounds violation
  - Protection error
    - e.g., write to read-only file
  - Arithmetic error
  - I/O failure
  - Invalid instruction
    - happens when trying to execute data
  - Privileged instruction
  - Operating system intervention (OS介入)
    - such as when deadlock occurs.
  - Parent request to terminate one child
  - Parent terminates so child processes terminate.

Operations on Processes

- **Process Termination**
  - Procedure of process termination
    - A process may execute last statement and ask the operating system to terminate it by exit() system call.
      - Its entry in the process table remains there until her parent, if exists, calls wait().
      - Its resources are deallocated by operating system.
    - Parent may terminate execution of child processes:
      - Child has exceeded allocated resources.
      - Mission assigned to child is no longer required.
      - If parent process is exiting:
        - Some OSes do not allow child to continue if its parent terminates.
        - Cascading termination (级联终止) – all children terminated.

Operations on Processes

Process Termination

- Procedure of process termination
  - Prototype of wait()

```
#include <sys/wait.h>
  /* pid_t wait(int *status); */

pid_t pid;
int status;
pid = wait(&status);
```

  - When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.

Process Termination

- Zombies and Orphans
  - A process that has terminated, but whose parent has not yet called wait(), is defunct and known as a *zombie process* (僵尸进程).
    - All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
  - Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as *orphans* (孤儿进程).
  - Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes (*adoption* of orphans).
    - The init process is the root of the process hierarchy in UNIX and Linux systems.
    - The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Operations on Processes

Process Termination

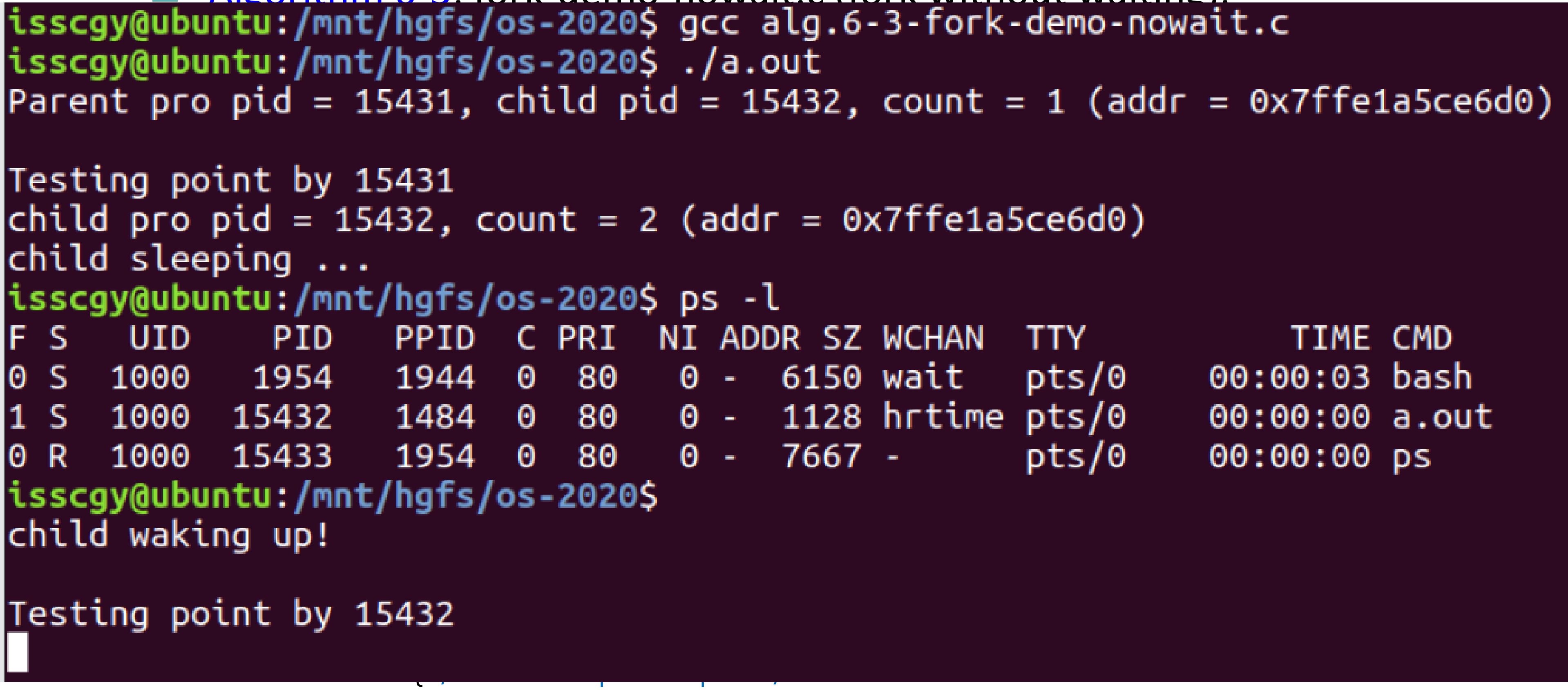
- Algorithm 6-3: fork-demo-nowait.c (fork without waiting).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
// #include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid;

    childpid = fork(); /* child duplicates parent’s address space */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    }
    else
        if (childpid == 0) { /* This is child pro */
            count++;
            printf("child pro pid = %d, count = %d (addr = %p)\n", getpid(), count,
&count);
            printf("child sleeping ...\n");
            sleep(10); /* parent exites during this period, child became an orphan */
            printf("\nchild waking up!\n");
        }
        else { /* This is parent pro */
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
        }
    printf("\nTesting point by %d\n", getpid()); /* executed by parent and child */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **Process Termination**
  - Algorithm 6-3: fork-demo-nowait.c (fork without waiting).

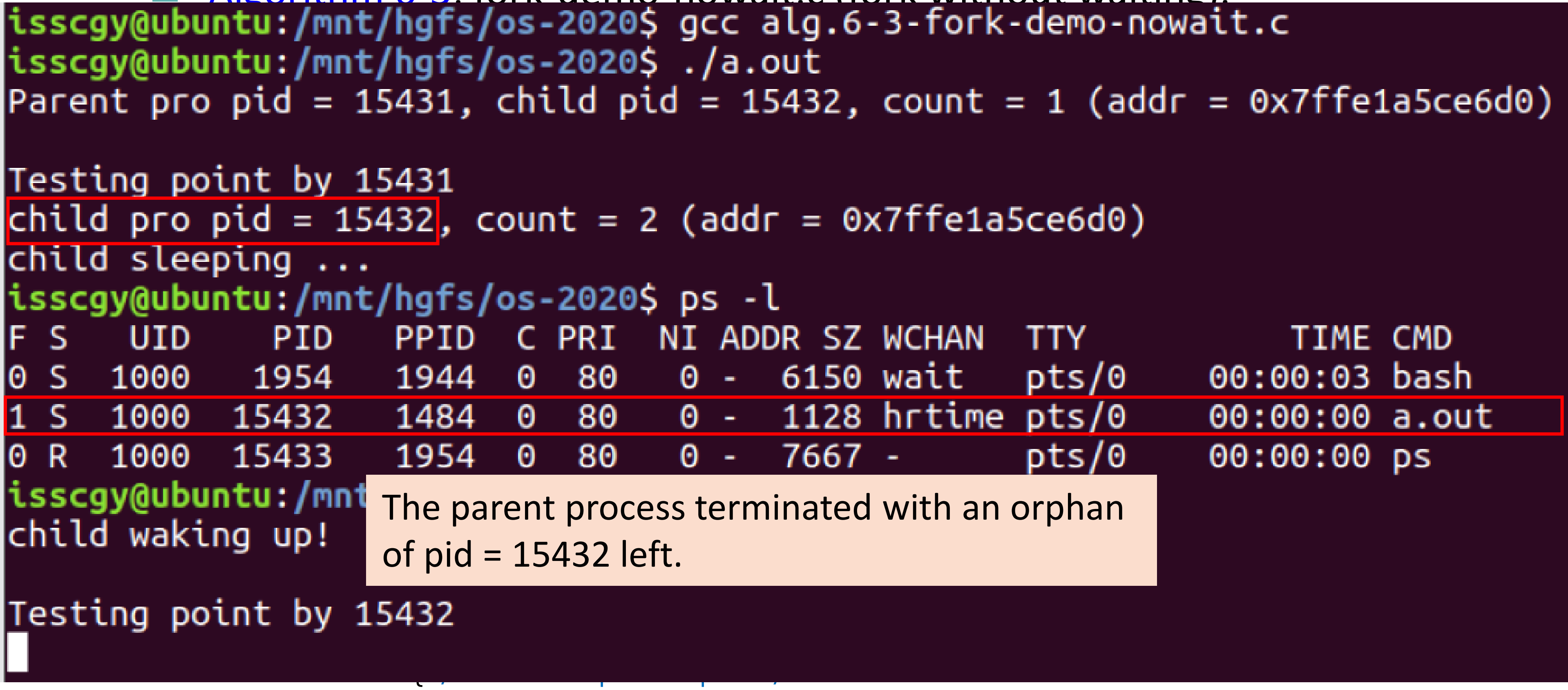


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-3-fork-demo-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 15431, child pid = 15432, count = 1 (addr = 0x7ffe1a5ce6d0)

Testing point by 15431
child pro pid = 15432, count = 2 (addr = 0x7ffe1a5ce6d0)
child sleeping ...
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
1 S  1000   15432    1484  0  80   0 -  1128 hrtime pts/0    00:00:00 a.out
0 R  1000   15433    1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
child waking up!

Testing point by 15432
```

```
                printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
            }
    printf("\nTesting point by %d\n", getpid()); /* executed by parent and child */
    return EXIT_SUCCESS;
}
```

Process Termination

- Algorithm 6-3: fork-demo-nowait.c (fork without waiting).



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-3-fork-demo-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 15431, child pid = 15432, count = 1 (addr = 0x7ffe1a5ce6d0)

Testing point by 15431
child pro pid = 15432, count = 2 (addr = 0x7ffe1a5ce6d0)
child sleeping ...
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
1 S  1000   15432    1484  0  80   0 -  1128 hrtime pts/0    00:00:00 a.out
0 R  1000   15433    1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt
child waking up!

Testing point by 15432
```

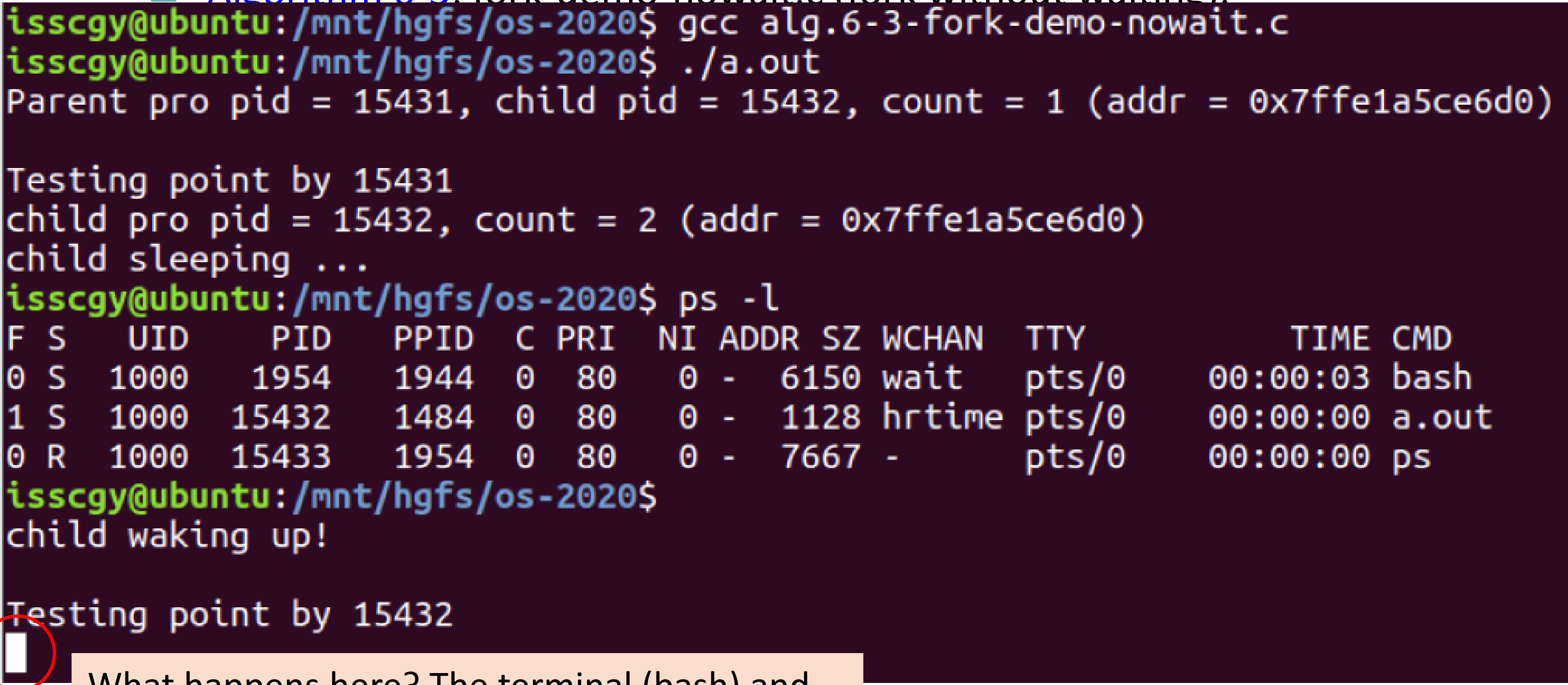
The parent process terminated with an orphan of pid = 15432 left.

```
            printf("parent pro pid = %d, child pid = %d, count = %d (addr = %p)\n",
getpid(), childpid, count, &count);
        }
    printf("\nTesting point by %d\n", getpid()); /* executed by parent and child */
    return EXIT_SUCCESS;
}
```

Operations on Processes

Process Termination

- Algorithm 6-3: fork-demo-nowait.c (fork without waiting).



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-3-fork-demo-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
Parent pro pid = 15431, child pid = 15432, count = 1 (addr = 0x7ffe1a5ce6d0)

Testing point by 15431
child pro pid = 15432, count = 2 (addr = 0x7ffe1a5ce6d0)
child sleeping ...
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI   NI ADDR SZ WCHAN   TTY          TIME CMD
0 S  1000    1954    1944  0  80    0 -  6150 wait    pts/0    00:00:03 bash
1 S  1000   15432    1484  0  80    0 -  1128 hrtime  pts/0    00:00:00 a.out
0 R  1000   15433    1954  0  80    0 -  7667 -       pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
child waking up!

Testing point by 15432
```

What happens here? The terminal (bash) and the forked child are working asynchronously.

```
ld pid = %d, count = %d (addr = %p)\n",
        }
        printf("\nTesting point by %d\n", getpid()); /* executed by parent and child */
        return EXIT_SUCCESS;
    }
```

Process Termination

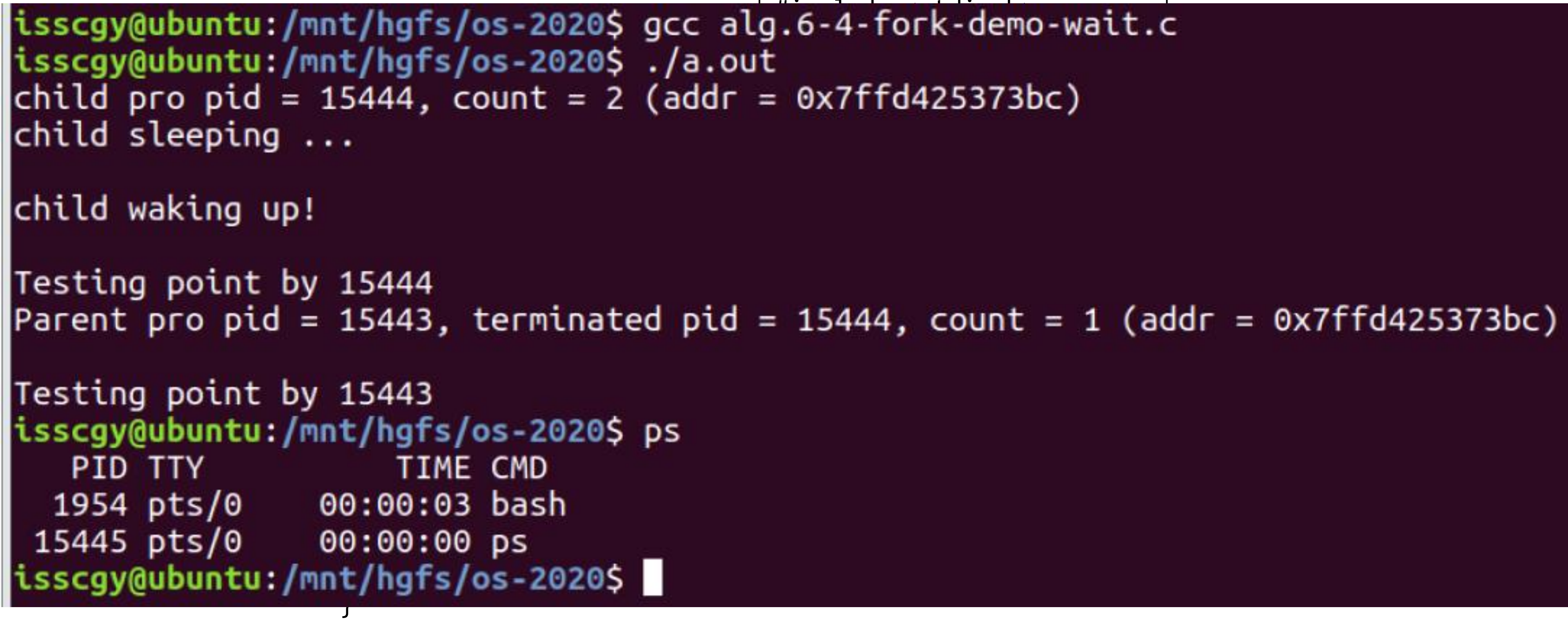
- Algorithm 6-4: fork-demo-wait.c (fork and wait).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void)
{
    int count = 1;
    pid_t childpid, terminatedid;

    childpid = fork(); /* child duplicates parent’s address space */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    }
    else
        if (childpid == 0) { /* This is child pro */
            count++;
            printf("child pro pid = %d, count = %d (addr = %p)\n", getpid(), count,
&count);
            printf("child sleeping ...\n");
            sleep(5); /* parent wait() during this period */
            printf("\nchild waking up!\n");
        }
        else { /* This is parent pro */
            terminatedid = wait(0);
            printf("parent pro pid = %d, terminated pid = %d, count = %d (addr =
%p)\n", getpid(), terminatedid, count, &count);
        }
    printf("\nTesting point by %d\n", getpid()); /* executed by child and parent */
    return EXIT_SUCCESS;
}
```

Process Termination

- Algorithm 6-4: fork-demo-wait.c (fork and wait).

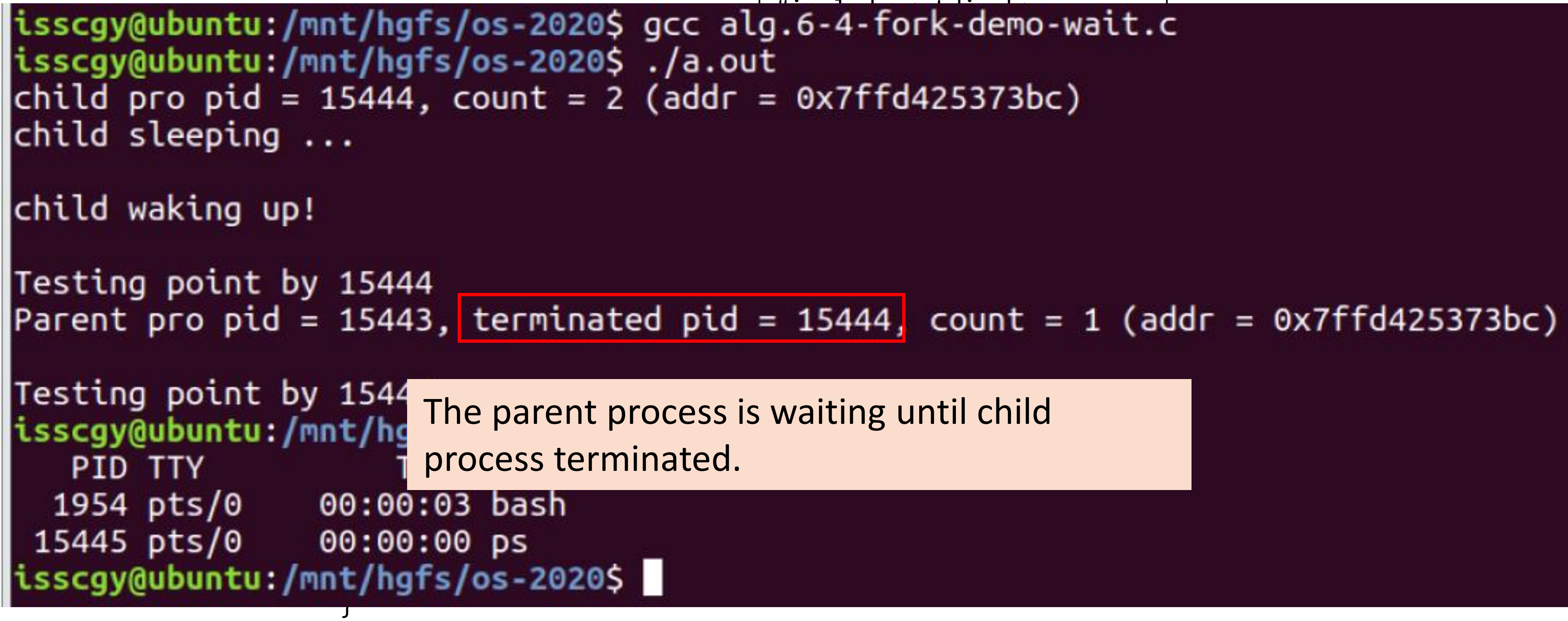


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-4-fork-demo-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
child pro pid = 15444, count = 2 (addr = 0x7ffd425373bc)
child sleeping ...

child waking up!

Testing point by 15444
Parent pro pid = 15443, terminated pid = 15444, count = 1 (addr = 0x7ffd425373bc)

Testing point by 15443
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps
   PID TTY          TIME CMD
  1954 pts/0    00:00:03 bash
 15445 pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ 
```

```
        else { /* This is parent pro */
            terminatedid = wait(0);
            printf("parent pro pid = %d, terminated pid = %d, count = %d (addr =
%p)\n", getpid(), terminatedid, count, &count);
        }
    printf("\nTesting point by %d\n", getpid()); /* executed by child and parent */
    return EXIT_SUCCESS;
}
```

Process Termination

- Algorithm 6-4: fork-demo-wait.c (fork and wait).



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-4-fork-demo-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
child pro pid = 15444, count = 2 (addr = 0x7ffd425373bc)
child sleeping ...

child waking up!

Testing point by 15444
Parent pro pid = 15443, terminated pid = 15444, count = 1 (addr = 0x7ffd425373bc)

Testing point by 1544
isscgy@ubuntu:/mnt/hg
   PID TTY          T
  1954 pts/0    00:00:03 bash
 15445 pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ 
```

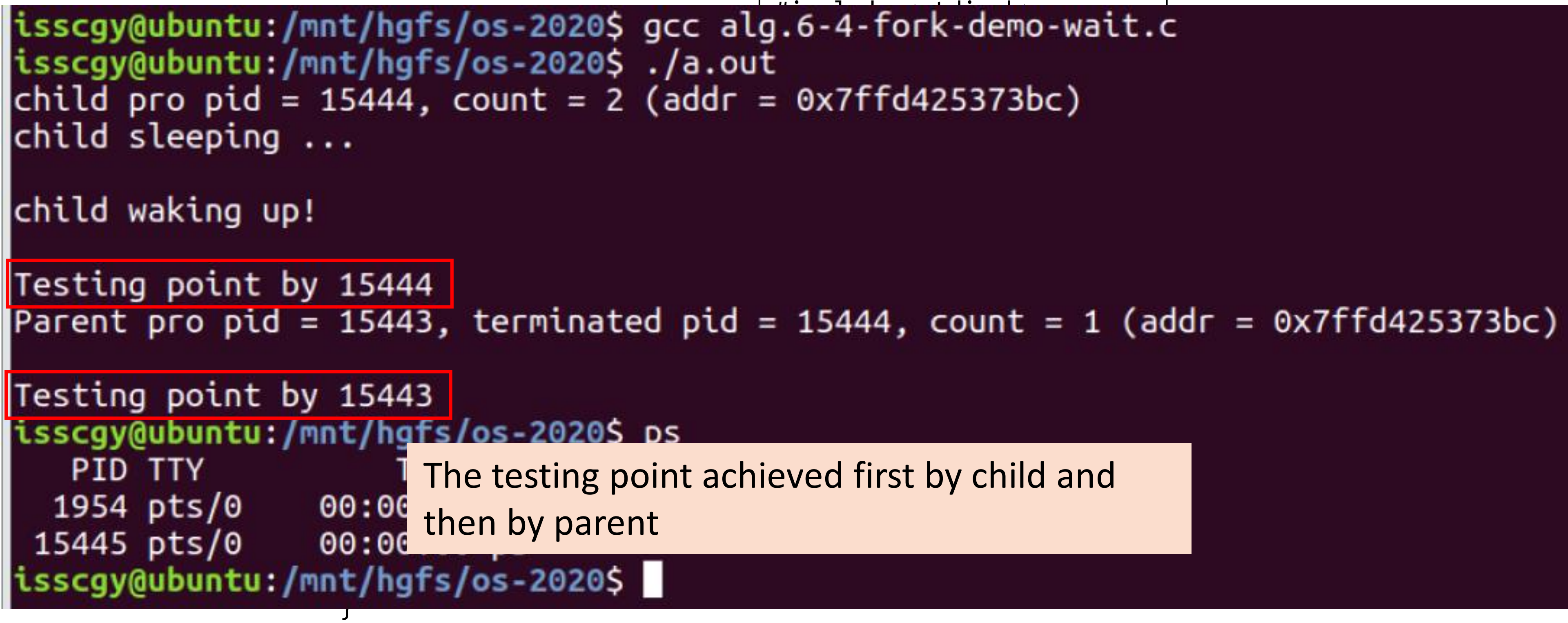
The parent process is waiting until child process terminated.

```
        else { /* This is parent pro */
            terminatedid = wait(0);
            printf("parent pro pid = %d, terminated pid = %d, count = %d (addr = 
%p)\n", getpid(), terminatedid, count, &count);
        }
    printf("\nTesting point by %d\n", getpid()); /* executed by child and parent */
    return EXIT_SUCCESS;
}
```

Operations on Processes

Process Termination

- Algorithm 6-4: fork-demo-wait.c (fork and wait).



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-4-fork-demo-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out
child pro pid = 15444, count = 2 (addr = 0x7ffd425373bc)
child sleeping ...

child waking up!

Testing point by 15444
Parent pro pid = 15443, terminated pid = 15444, count = 1 (addr = 0x7ffd425373bc)

Testing point by 15443
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps
   PID TTY          T
  1954 pts/0    00:00
 15445 pts/0    00:00
isscgy@ubuntu:/mnt/hgfs/os-2020$ 
```

The testing point achieved first by child and then by parent

```
        else { /* This is parent pro */
            terminatedid = wait(0);
            printf("parent pro pid = %d, terminated pid = %d, count = %d (addr =
%p)\n", getpid(), terminatedid, count, &count);
        }
    printf("\nTesting point by %d\n", getpid()); /* executed by child and parent */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **Process Termination**
  - Algorithm 6-5-0: sleeper.c (a demo process sleeping for 5 seconds).

```
/* gcc -o alg.6-5-0-sleeper.o alg.6-5-0-sleeper.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    int secnd = 5;

    if (argc > 1) {
        secnd = atoi(argv[1]);
        if ( secnd <= 0 || secnd > 10)
            secnd = 5;
    }

    printf("\nsleeper pid = %d, ppid = %d\nsleeper is taking a nap for %d 
seconds\n", getpid(), getppid(), secnd); /* ppid - its parent pro id */

    sleep(secnd);
    printf("\nsleeper wakes up and returns\n");

    return 0;
}
```

Process Termination

Algorithm 6-5: vfork-execv-wait.c (vfork, execv and wait) (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <wait.h>
```

```
int main(void)
{
    pid_t childpid;

    childpid = vfork();
        /* child shares parent's address space */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    }
    else
        if(childpid == 0) { /* This is child pro */
            printf("This is child, pid = %d, taking a nap for 2 sencods \n", getpid());
            sleep(2); /* parent hung up and do nothing */

            char filename[80];
            struct stat buf;
            strcpy(filename, "./alg.6-5-0-sleeper.o");
            if(stat(filename, &buf) == -1) {
                perror("\nsleeper stat()");
                _exit(0);
            }
            char *argv1[] = {filename, argv[1], NULL};
            printf("child waking up and again execv() a sleeper: %s %s\n\n", argv1[0], 
argv1[1]);
            execv(filename, argv); /* parent resume at the point 'execv' called */
        }
```

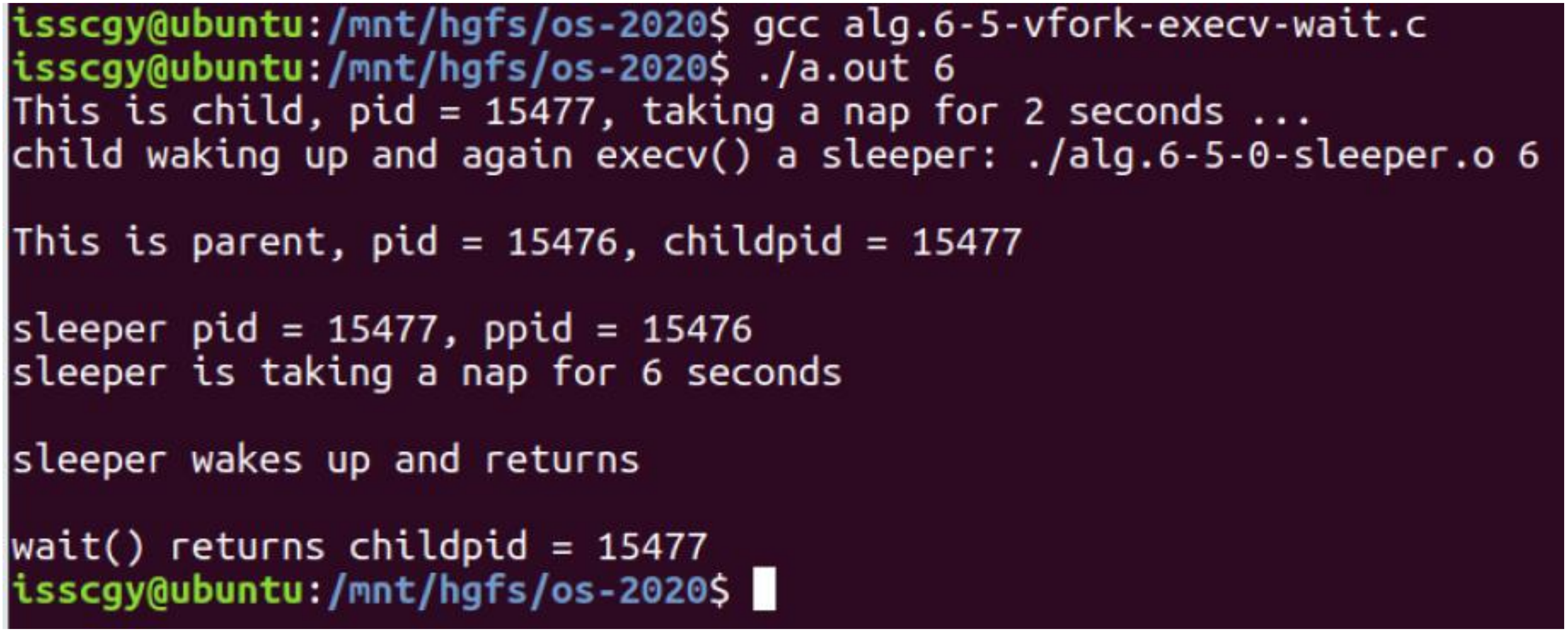
Operations on Processes

- **Process Termination**
  - Algorithm 6-5: vfork-execv-wait.c (vfork, execv and wait) (2)

```
        else { /* This is parent pro, start when the vforked child terminated */
            printf("This is parent, pid = %d, childpid = %d \n", getpid(), childpid);
/* parent executed this statement during the EXECV time */
            int retpid = wait(0); /* without wait(), the spawned EXECV may became an 
orphan */
            printf("\nwait() returns childpid = %d\n", retpid);
        }

    return EXIT_SUCCESS;
}
```

Operations on Processes

- **Process Termination**
  - Algorithm 6-5: vfork-execv-wait.c (vfork, execv and wait) (2)

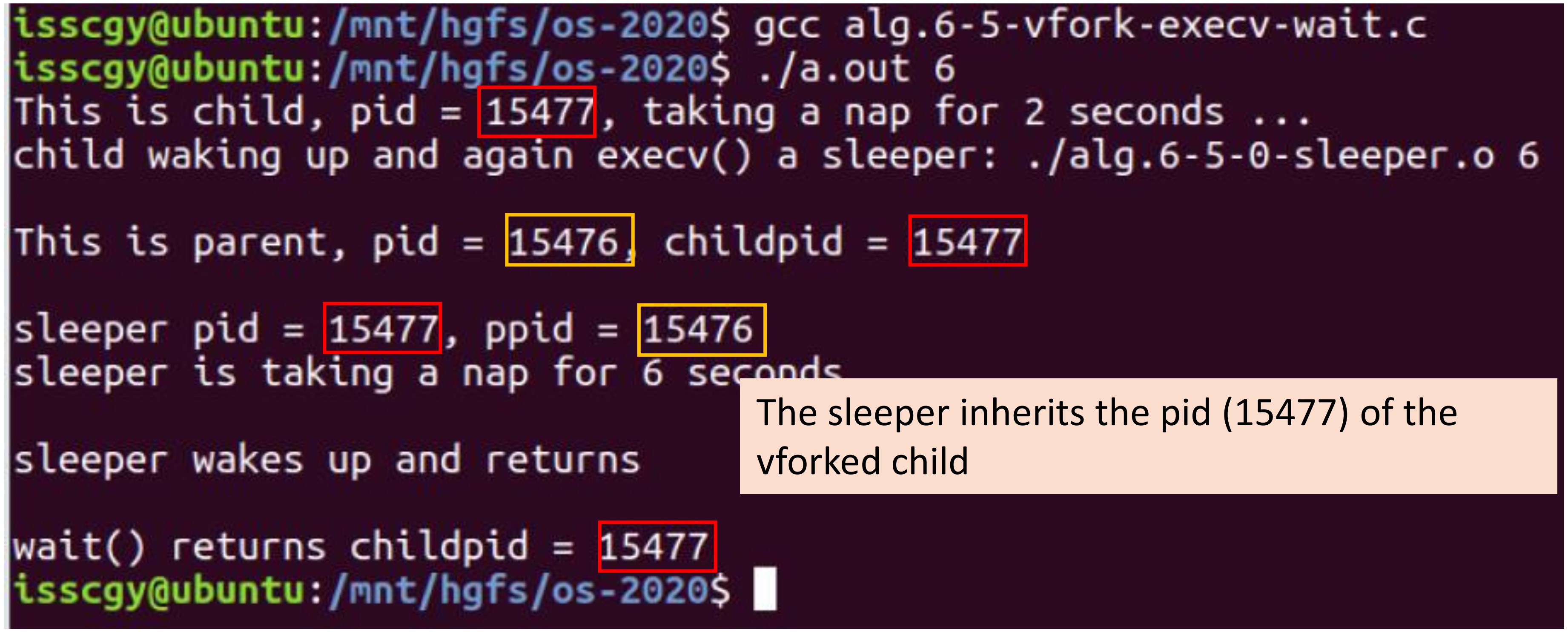


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 15477, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 15476, childpid = 15477

sleeper pid = 15477, ppid = 15476
sleeper is taking a nap for 6 seconds

sleeper wakes up and returns

wait() returns childpid = 15477
isscgy@ubuntu:/mnt/hgfs/os-2020$ █
```

Operations on Processes

Process Termination

- Algorithm 6-5: vfork-execv-wait.c (vfork, execv and wait) (2)



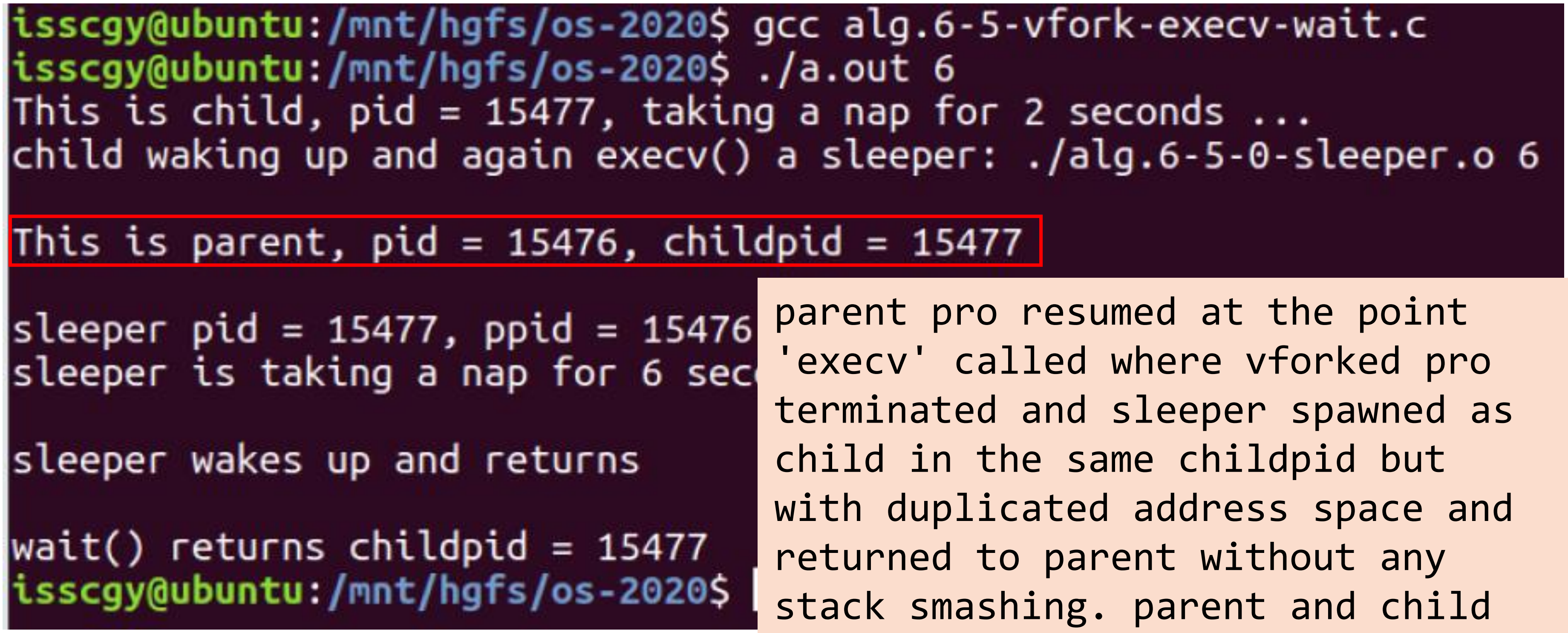
```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 15477, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 15476, childpid = 15477

sleeper pid = 15477, ppid = 15476
sleeper is taking a nap for 6 seconds

sleeper wakes up and returns

wait() returns childpid = 15477
isscgy@ubuntu:/mnt/hgfs/os-2020$ 
```

The sleeper inherits the pid (15477) of the vforked child

Operations on Processes

Process Termination

- Algorithm 6-5: vfork-execv-wait.c (vfork, execv and wait) (2)

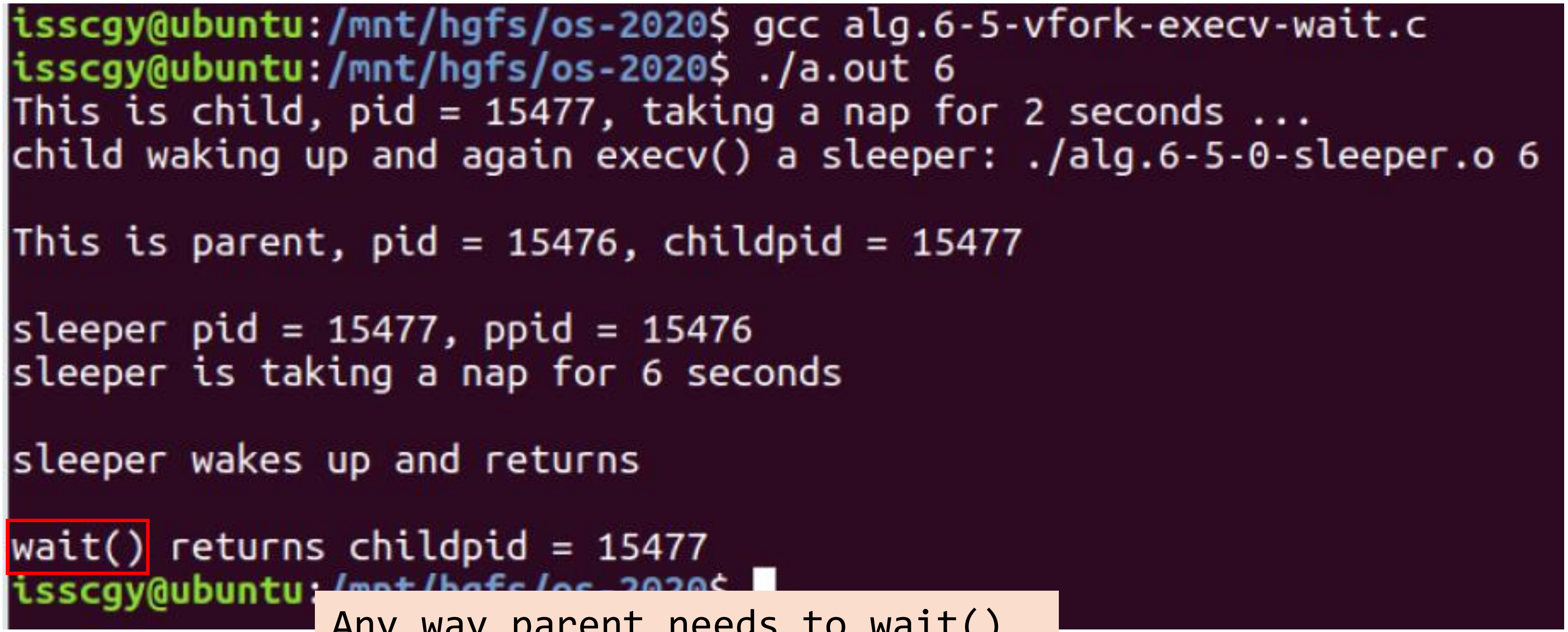


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 15477, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 15476, childpid = 15477

sleeper pid = 15477, ppid = 15476
sleeper is taking a nap for 6 sec

sleeper wakes up and returns

wait() returns childpid = 15477
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

```
parent pro resumed at the point
'execv' called where vforked pro
terminated and sleeper spawned as
child in the same childpid but
with duplicated address space and
returned to parent without any
stack smashing. parent and child
executed asynchronously.
```

Process Termination

- Algorithm 6-5: vfork-execv-wait.c (vfork, execv and wait) (2)



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 15477, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 15476, childpid = 15477

sleeper pid = 15477, ppid = 15476
sleeper is taking a nap for 6 seconds

sleeper wakes up and returns

wait() returns childpid = 15477
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

Any way parent needs to wait() his children, or the spawned sleeper pro may become an orphan

Operations on Processes

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <wait.h>
```

```
int main(int argc, char* argv[])
{
    pid_t childpid;

    childpid = vfork();
        /* child shares parent's address space */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    }
    else
        if (childpid == 0) { /* This is child pro */
            printf("This is child, pid = %d, taking a nap for 2 seconds ... \n", 
getpid());
            sleep(2); /* parent hung up and do nothing */

            char filename[80];
            struct stat buf;
            strcpy(filename, "./alg.6-5-0-sleeper.o");
            if(stat(filename, &buf) == -1) {
                perror("\nsleeper stat()");
                _exit(0);
            }
            char *argv1[] = {filename, argv[1], NULL};
            printf("child waking up and again execv() a sleeper: %s %s\n\n", argv1[0], 
argv1[1]);
            execv(filename, argv); /* parent resume at the point 'execv' called */
        }
```

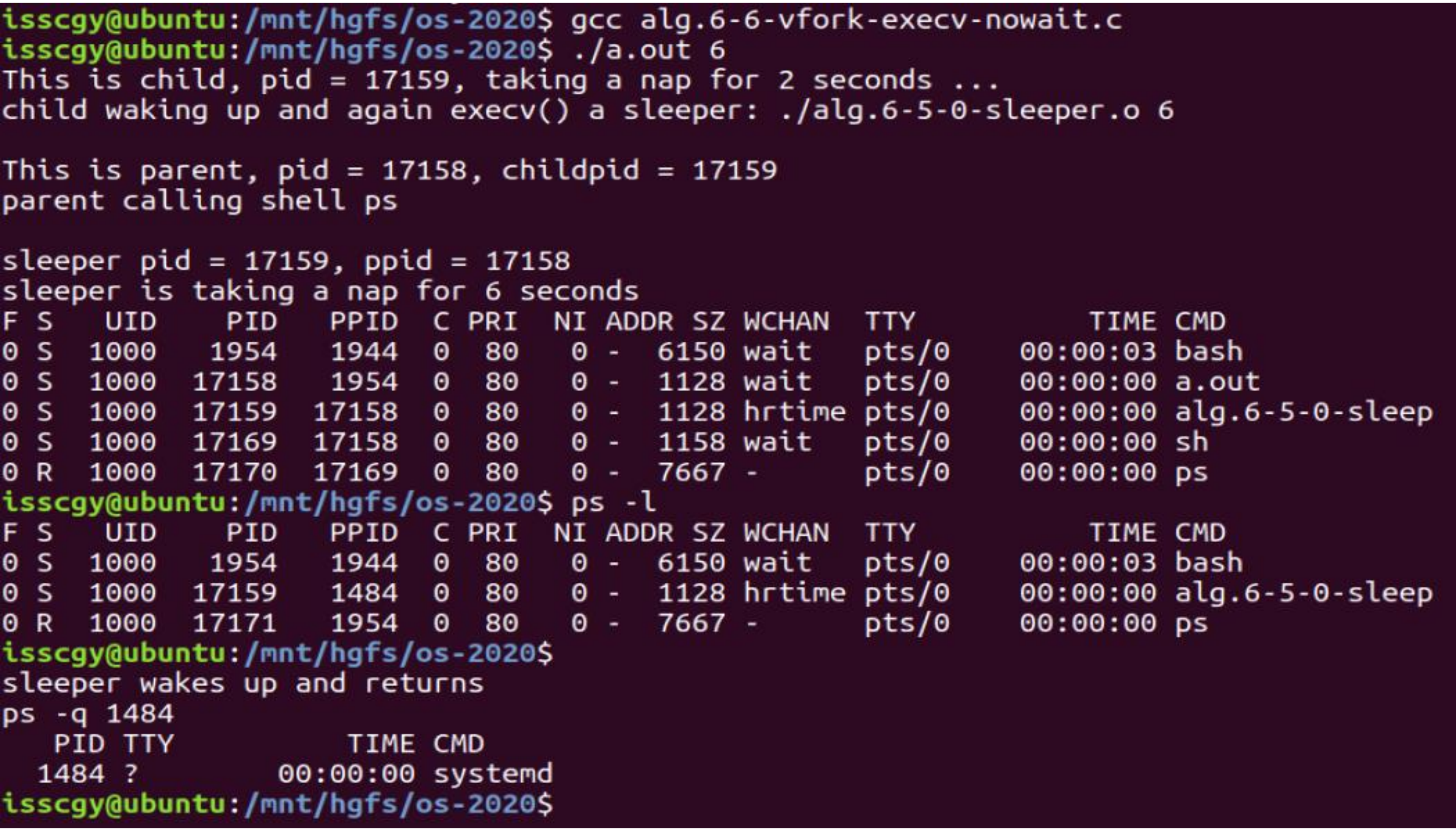
Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

```
        else { /* This is parent pro, start when the vforked child terminated */
                printf("This is parent, pid = %d, childpid = %d \n",getpid(), childpid);
                    /* parent executed this statement during the EXECV time */
                printf("parent calling shell ps\n");
                system("ps -l");
                sleep(1);
                return EXIT_SUCCESS;
                    /* parent exits without wait() and child may become an orphan */
        }
    }
```

Operations on Processes

- **Process Termination**
  - Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

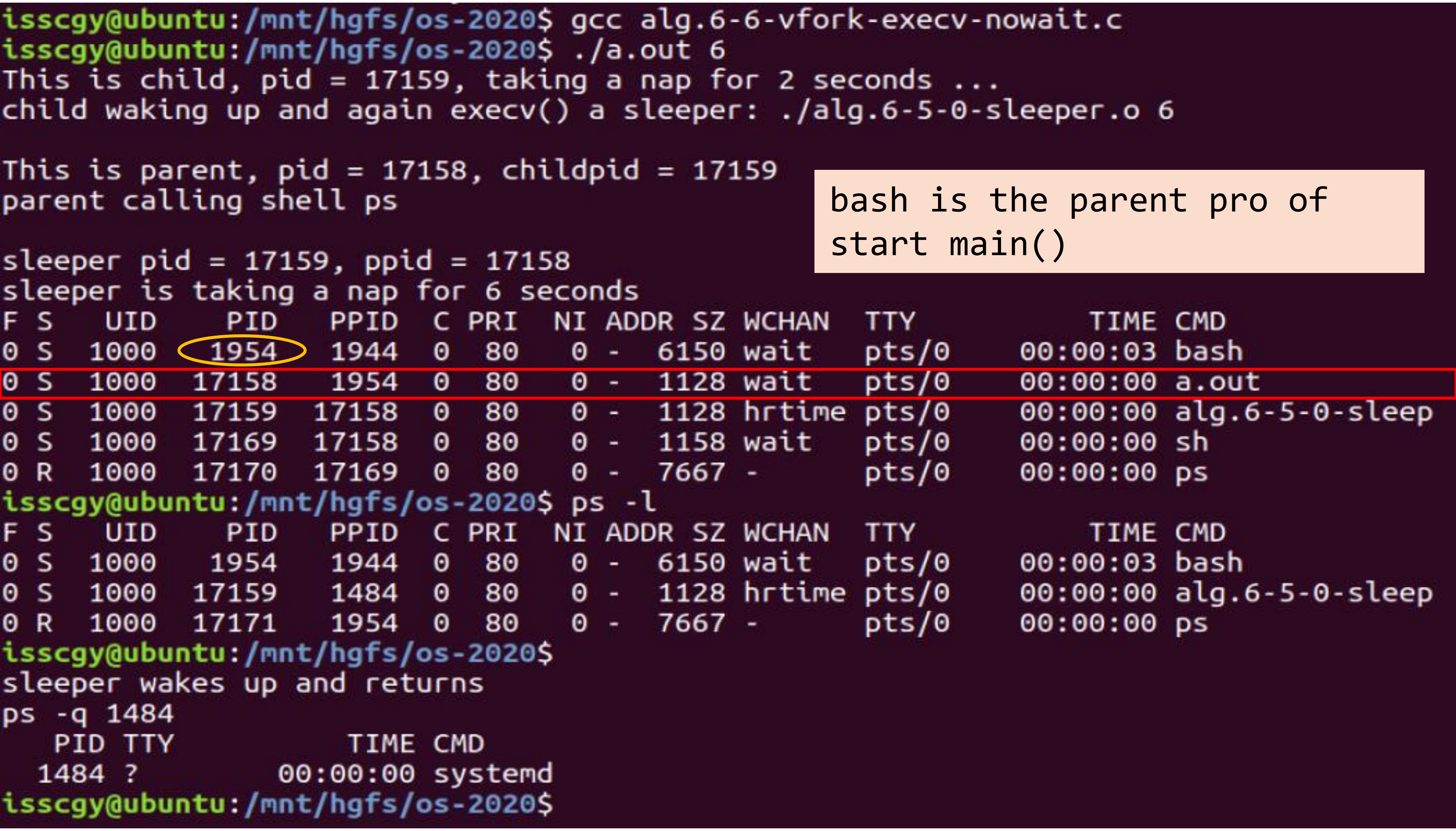


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17158    1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000   17159   17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000   17169   17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000   17170   17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17159    1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000   17171    1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
   PID TTY          TIME CMD
  1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

Operations on Processes

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

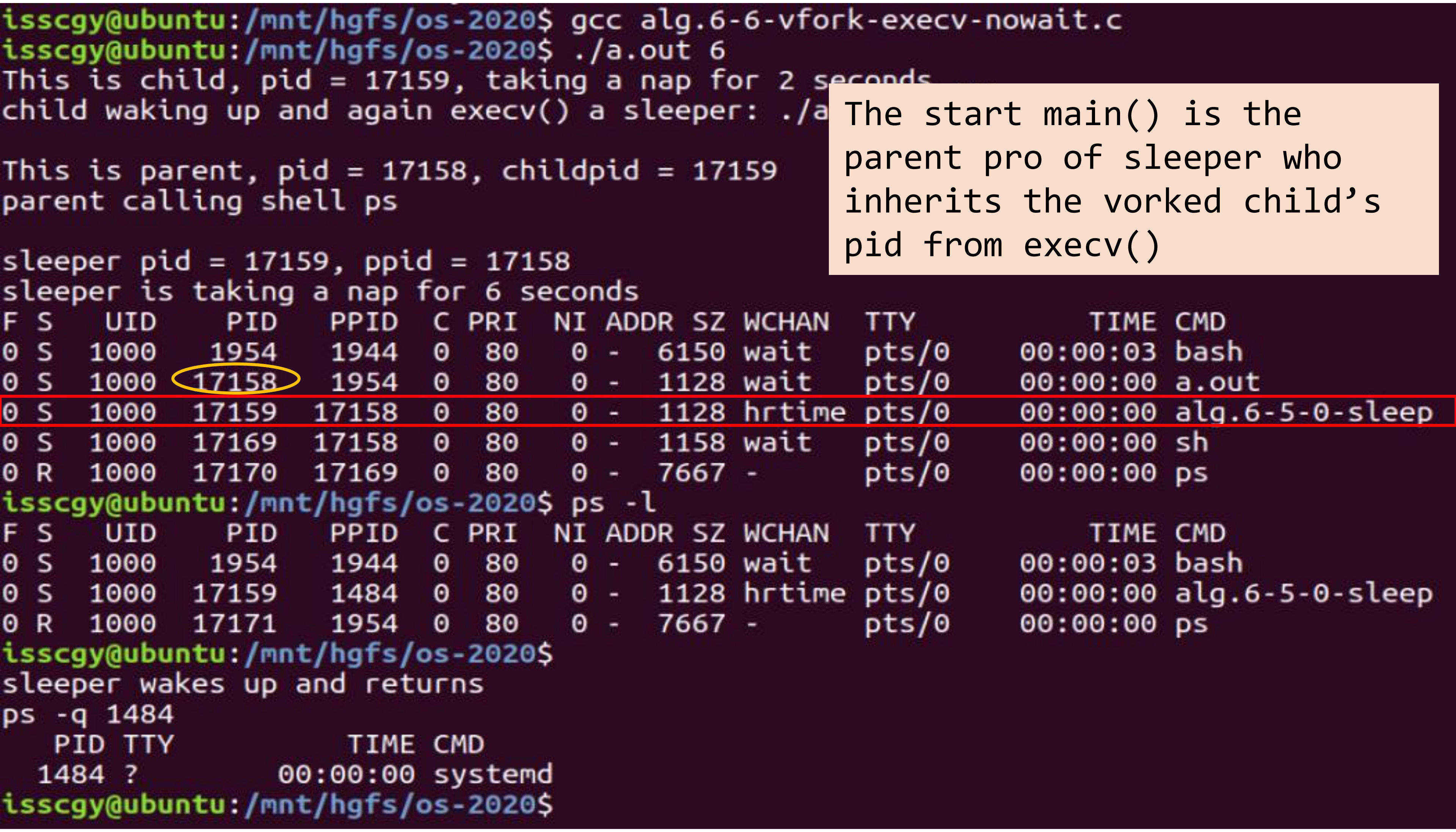


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17158   1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000  17159  17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000  17169  17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000  17170  17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17159   1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000  17171   1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
   PID TTY          TIME CMD
  1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

bash is the parent pro of start main()

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

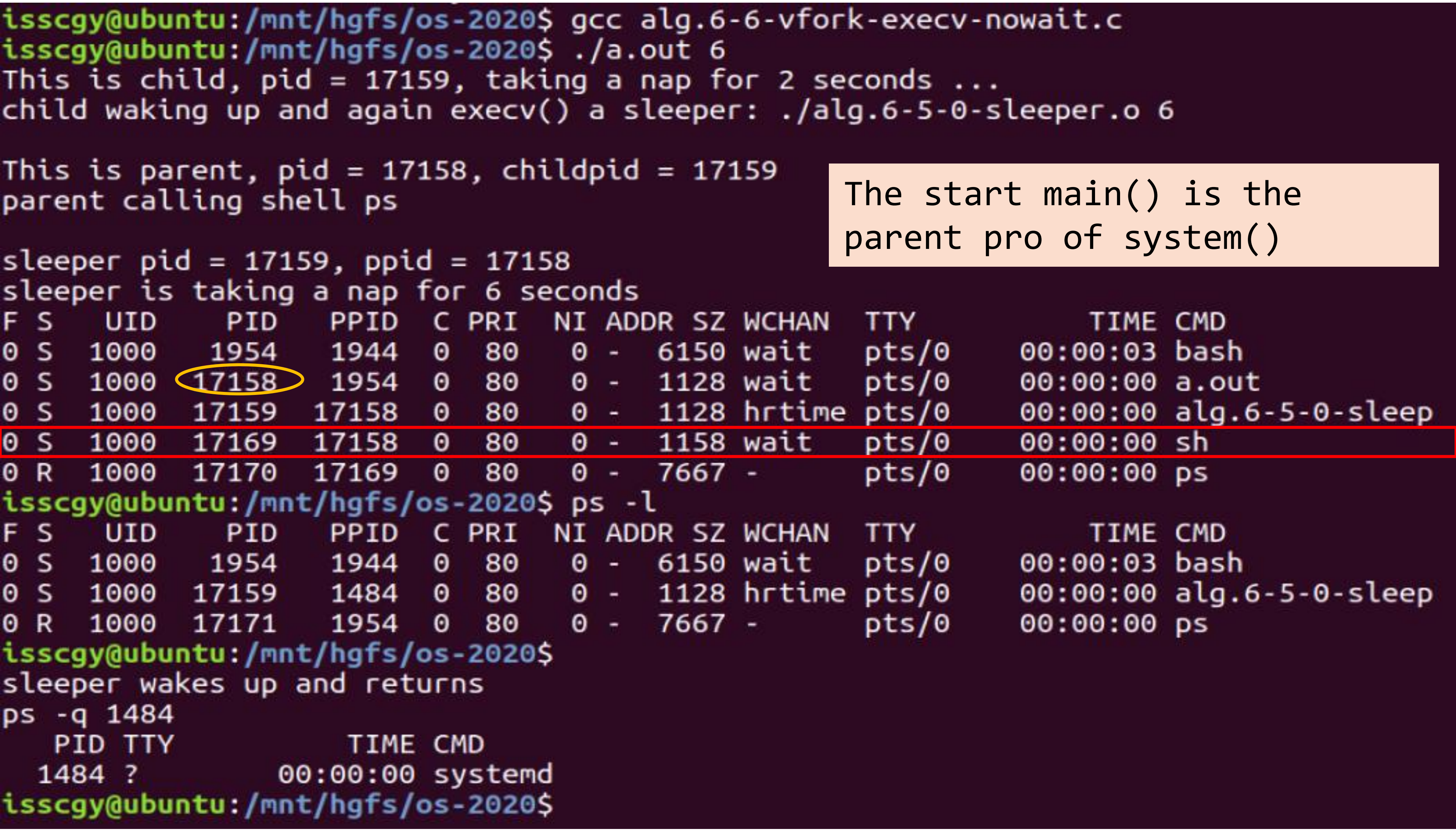


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds
child waking up and again execv() a sleeper: ./a

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17158    1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000   17159   17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000   17169   17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000   17170   17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17159    1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000   17171    1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
   PID TTY          TIME CMD
  1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

The start main() is the parent pro of sleeper who inherits the vorked child's pid from execv()

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

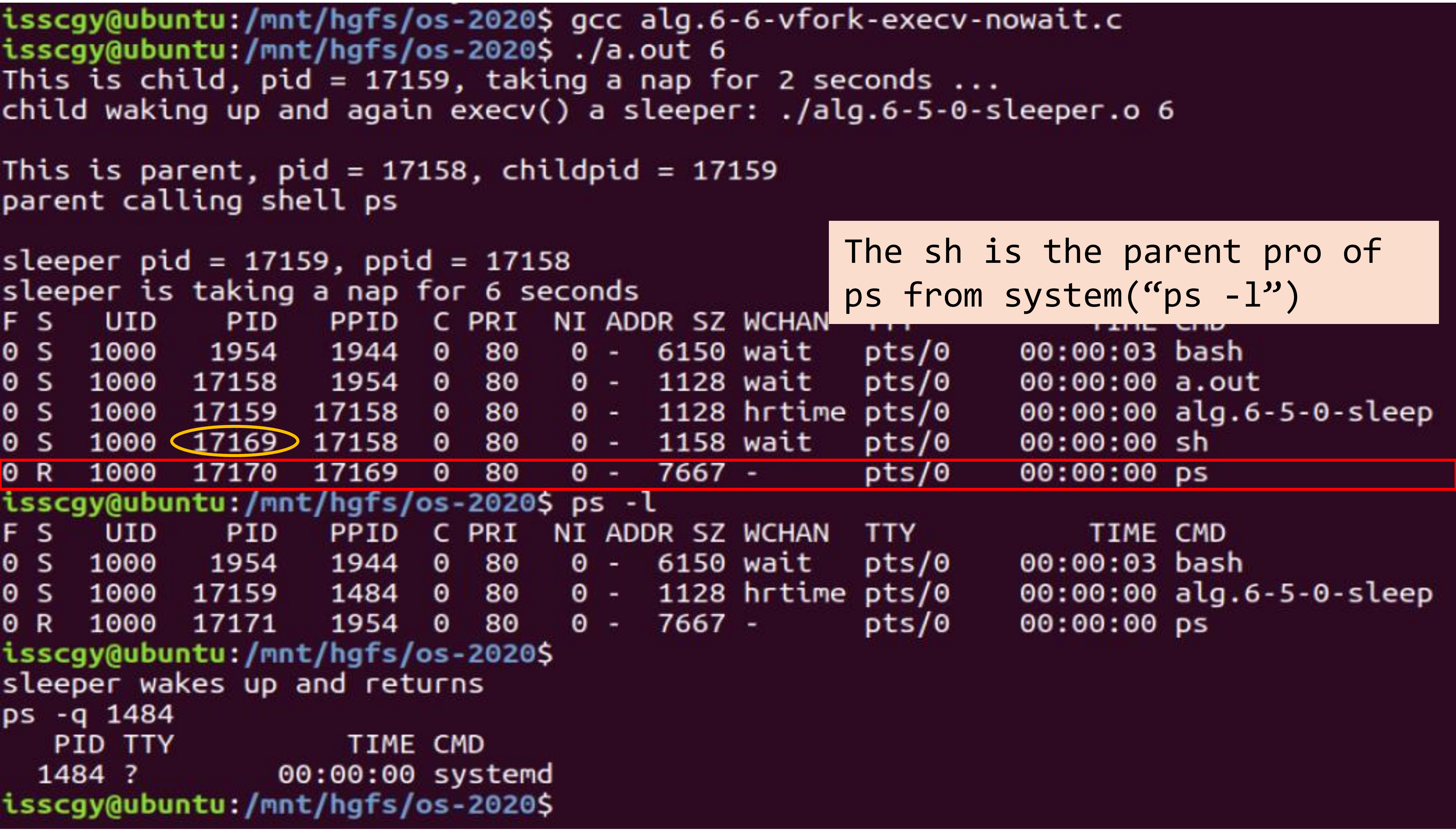


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17158    1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000   17159   17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000   17169   17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000   17170   17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17159    1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000   17171    1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
  PID TTY          TIME CMD
 1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

The start main() is the parent pro of system()

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)



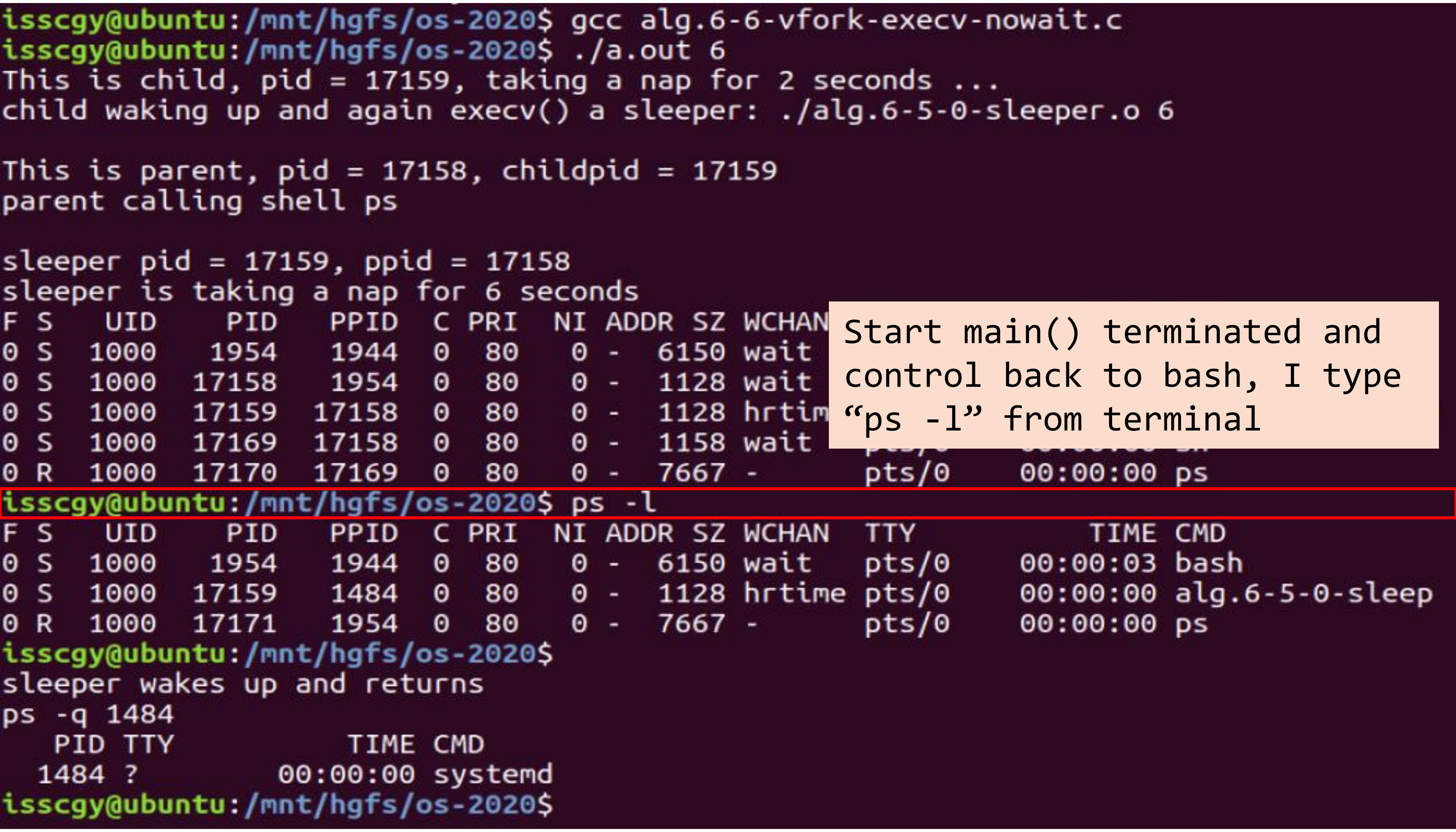
```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17158   1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000  17159  17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000  17169  17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000  17170  17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17159   1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000  17171   1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
   PID TTY          TIME CMD
  1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

The sh is the parent pro of ps from system("ps -l")

Operations on Processes

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

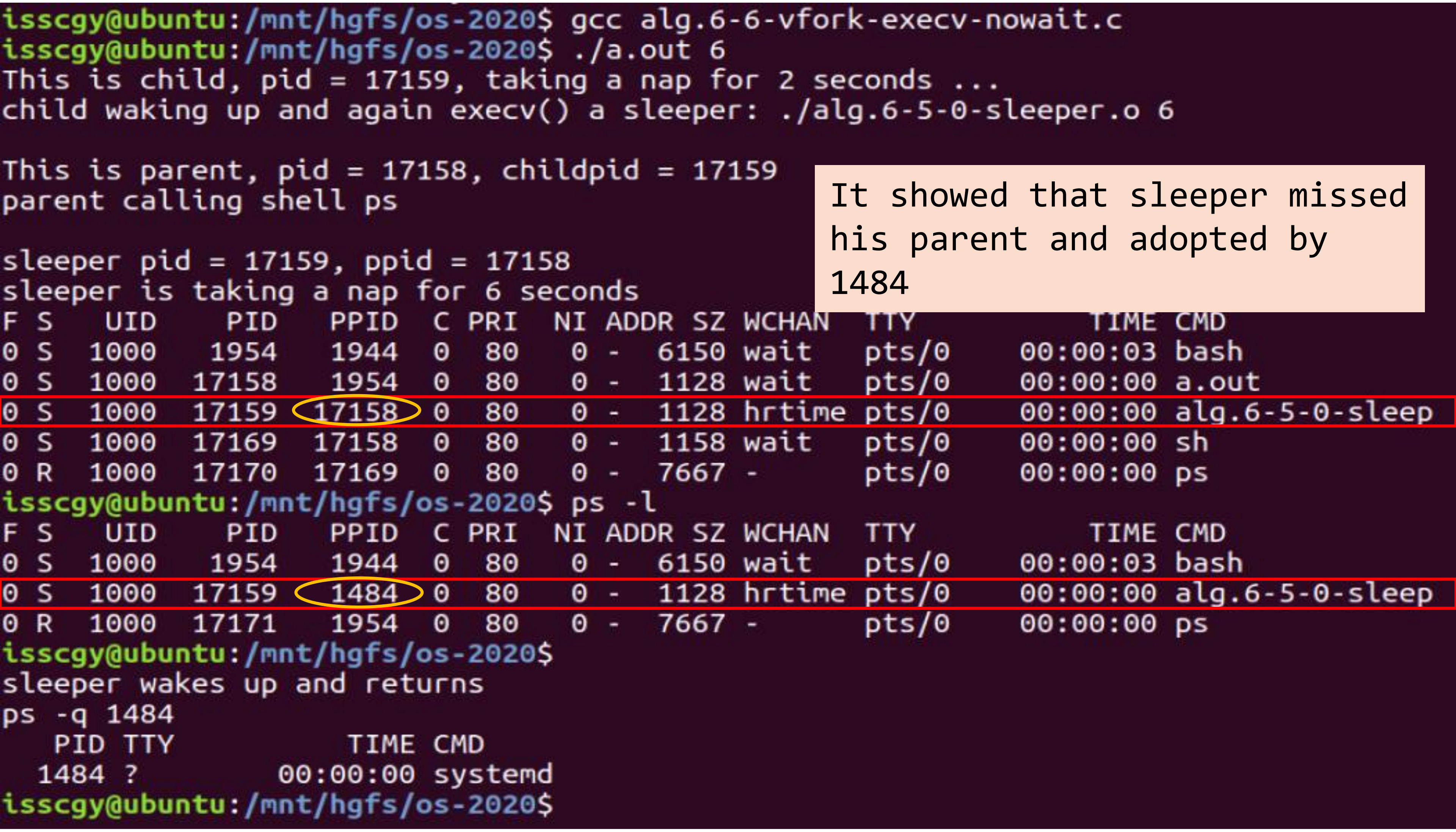


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN
0 S  1000   1954   1944  0  80   0 -  6150 wait
0 S  1000  17158   1954  0  80   0 -  1128 wait
0 S  1000  17159  17158  0  80   0 -  1128 hrtim
0 S  1000  17169  17158  0  80   0 -  1158 wait
0 R  1000  17170  17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17159   1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000  17171   1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
   PID TTY          TIME CMD
  1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

Start main() terminated and control back to bash, I type "ps -l" from terminal

Process Termination

- Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

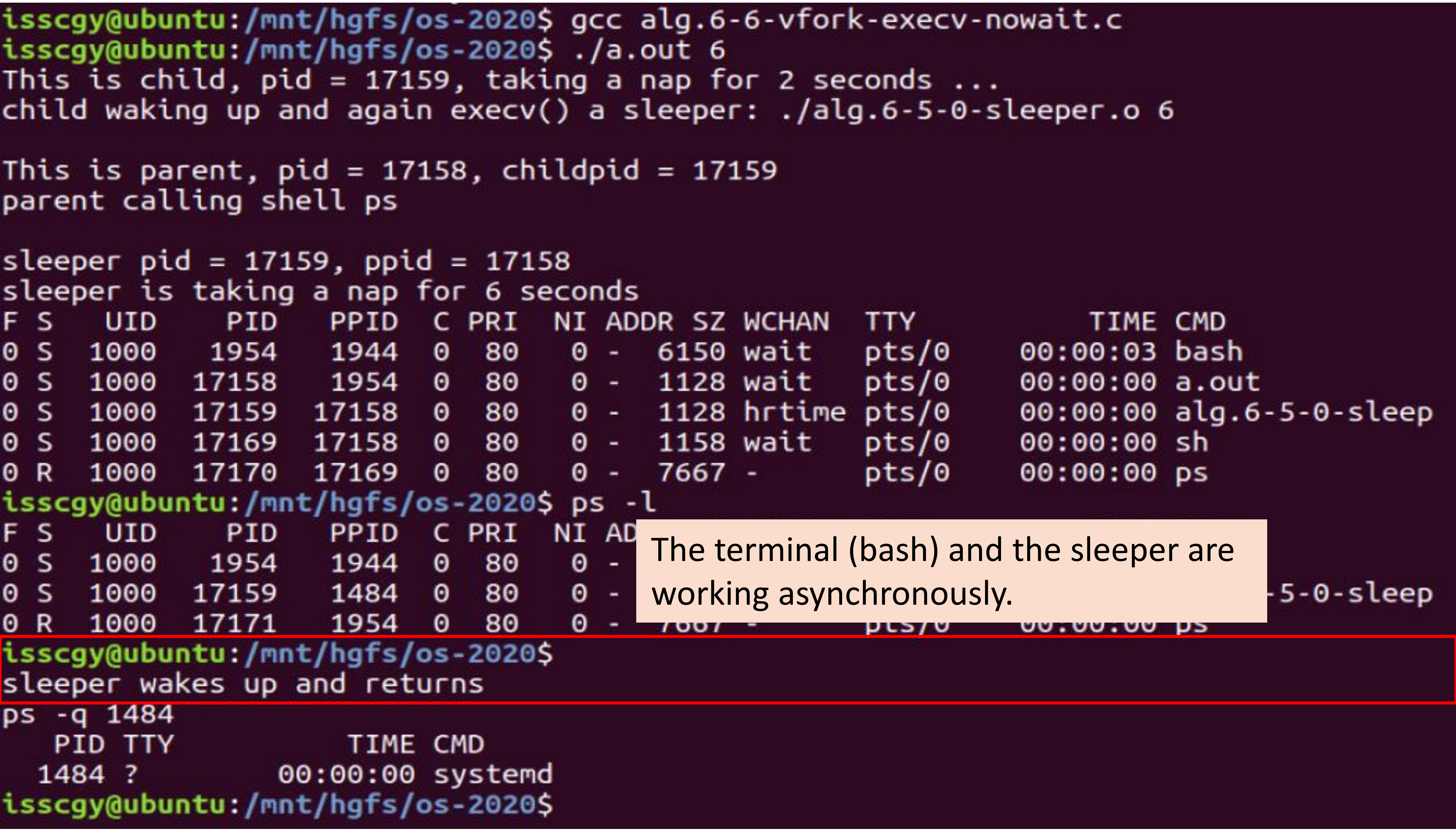


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1954  1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000 17158  1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000 17159 17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 17169 17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000 17170 17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1954  1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000 17159  1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 17171  1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
  PID TTY          TIME CMD
 1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

It showed that sleeper missed his parent and adopted by 1484

Operations on Processes

- **Process Termination**
  - Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)

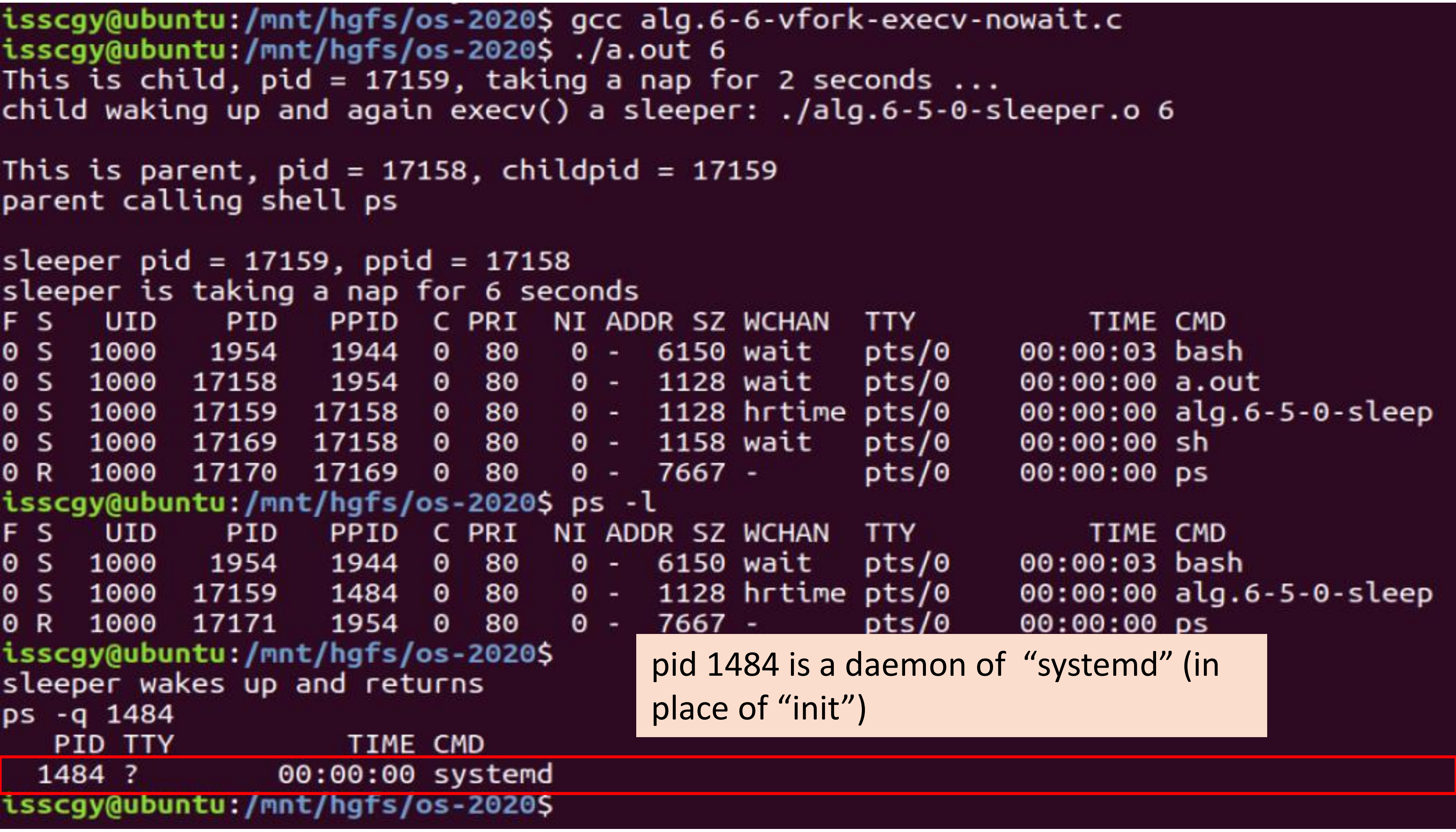


```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID     PID    PPID  C PRI   NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1954    1944  0  80    0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000   17158    1954  0  80    0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000   17159   17158  0  80    0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000   17169   17158  0  80    0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000   17170   17169  0  80    0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID     PID    PPID  C PRI   NI AD
0 S  1000    1954    1944  0  80    0 -
0 S  1000   17159    1484  0  80    0 -                                       -5-0-sleep
0 R  1000   17171    1954  0  80    0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
   PID TTY          TIME CMD
  1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

The terminal (bash) and the sleeper are working asynchronously.

Operations on Processes

- **Process Termination**
  - Algorithm 6-6: vfork-execv-nowait.c (vfork, execv without waiting) (2)



```
isscgy@ubuntu:/mnt/hgfs/os-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/os-2020$ ./a.out 6
This is child, pid = 17159, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0-sleeper.o 6

This is parent, pid = 17158, childpid = 17159
parent calling shell ps

sleeper pid = 17159, ppid = 17158
sleeper is taking a nap for 6 seconds
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17158   1954  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000  17159  17158  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000  17169  17158  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000  17170  17169  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   1954   1944  0  80   0 -  6150 wait   pts/0    00:00:03 bash
0 S  1000  17159   1484  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000  17171   1954  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/os-2020$
sleeper wakes up and returns
ps -q 1484
  PID TTY          TIME CMD
 1484 ?        00:00:00 systemd
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

pid 1484 is a daemon of “systemd” (in place of “init”)