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

Contents

Basic Concepts

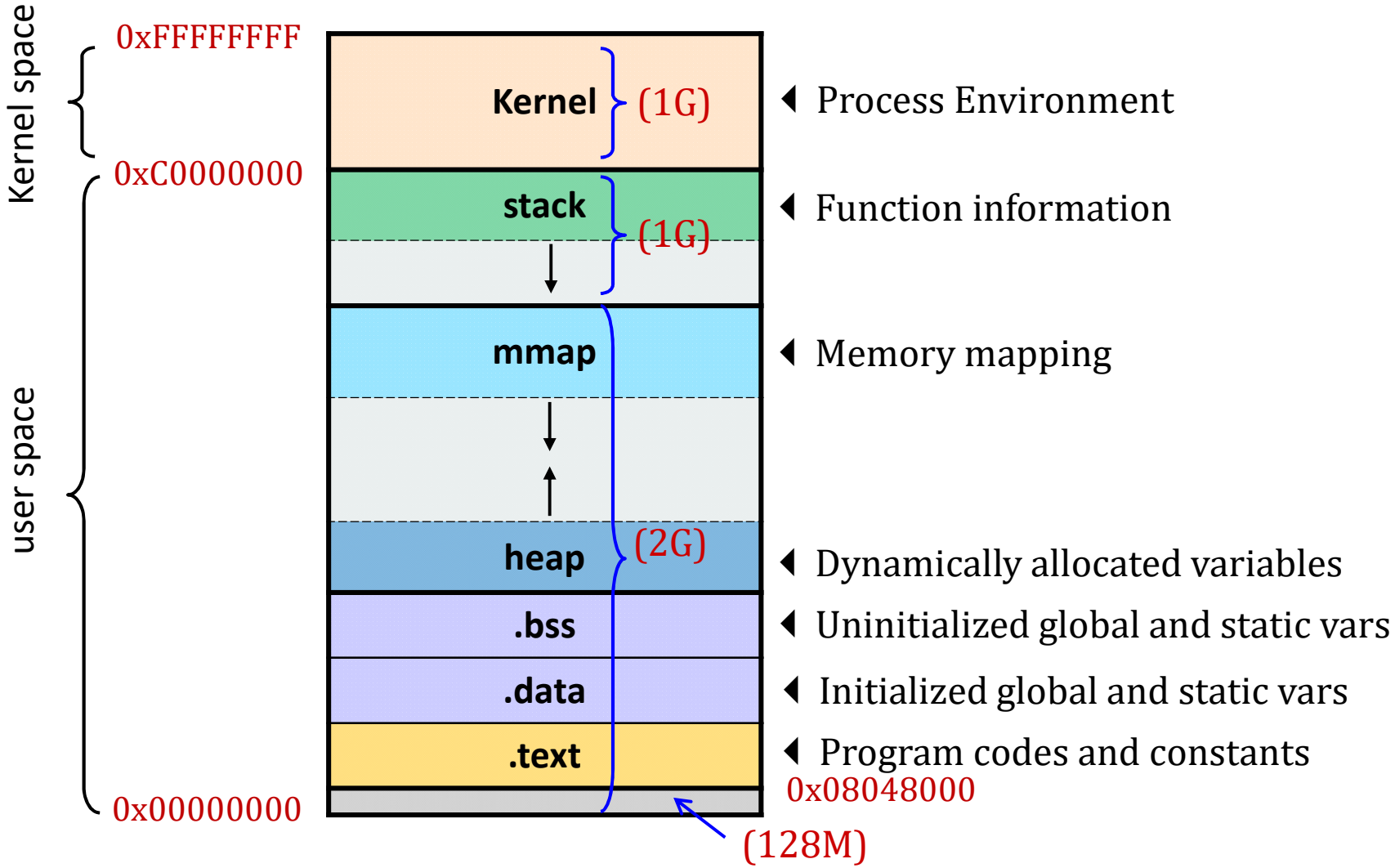
- Process is a program in execution; forms the basis of all computation. Process execution must progress in sequential fashion.
  - A program is a passive entity containing a list of instructions stored on disk as an executable file.
  - A process is an active entity with some specifications of the corresponding 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.

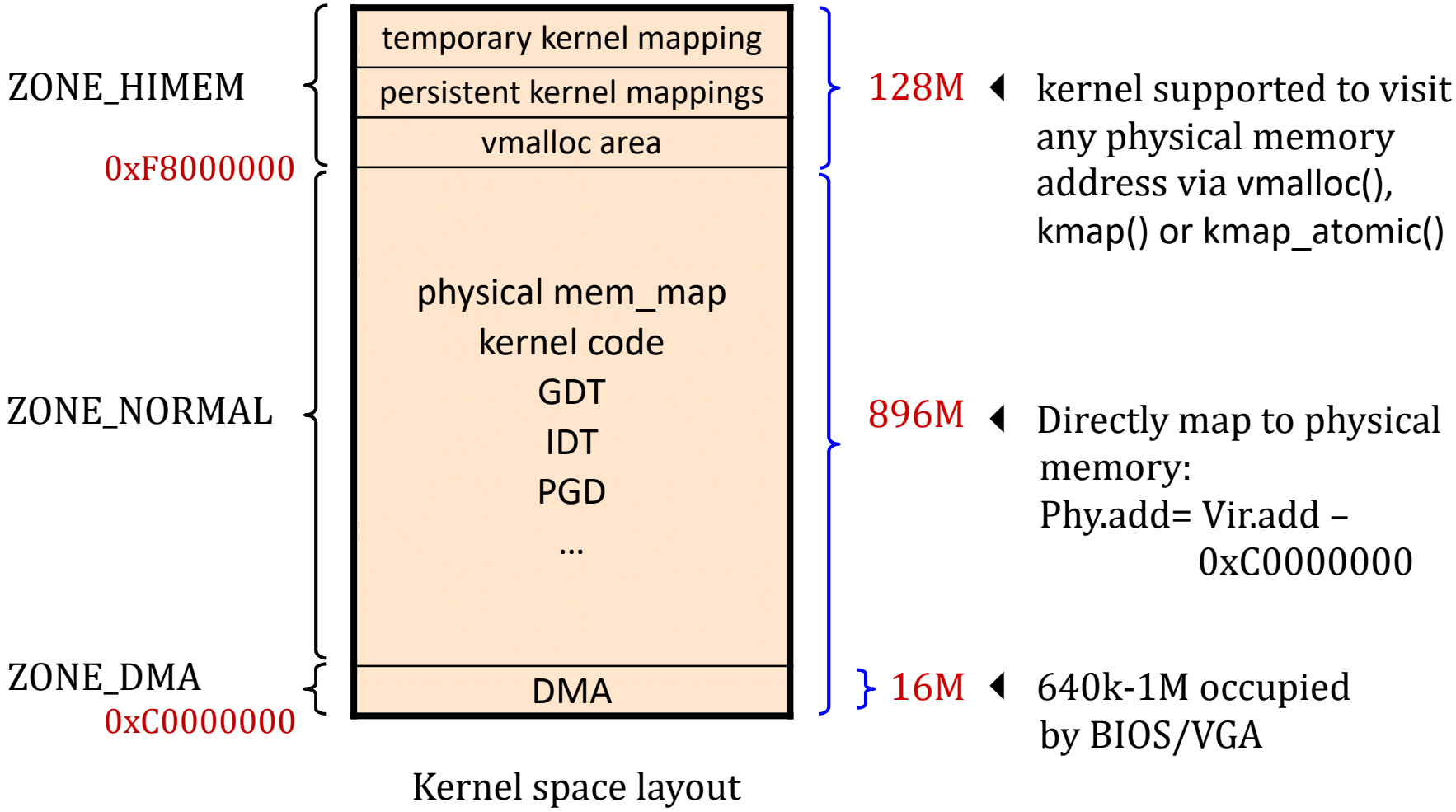
Basic Concepts

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



Basic Concepts

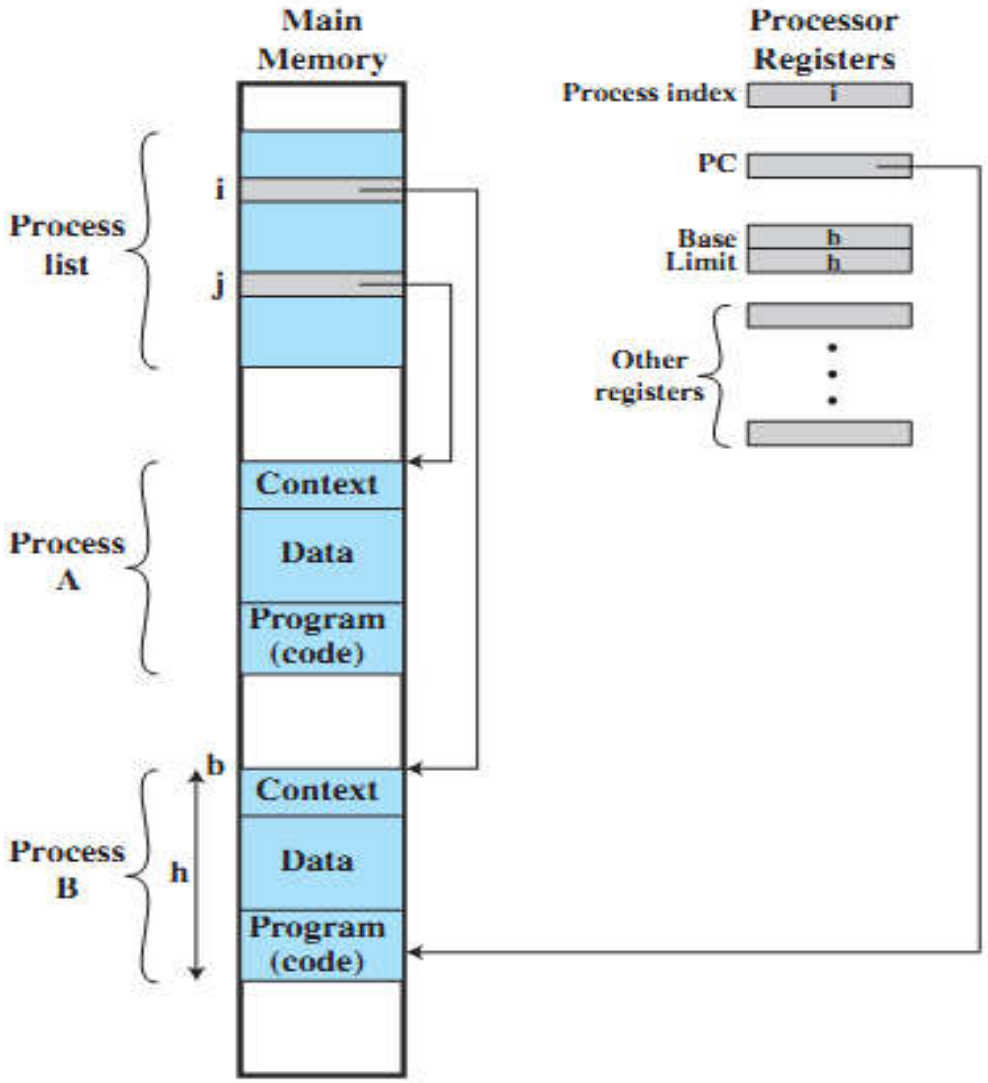
- Process Virtual Memory - Typical layout on Linux/IA-32.



Kernel space layout

Basic Concepts

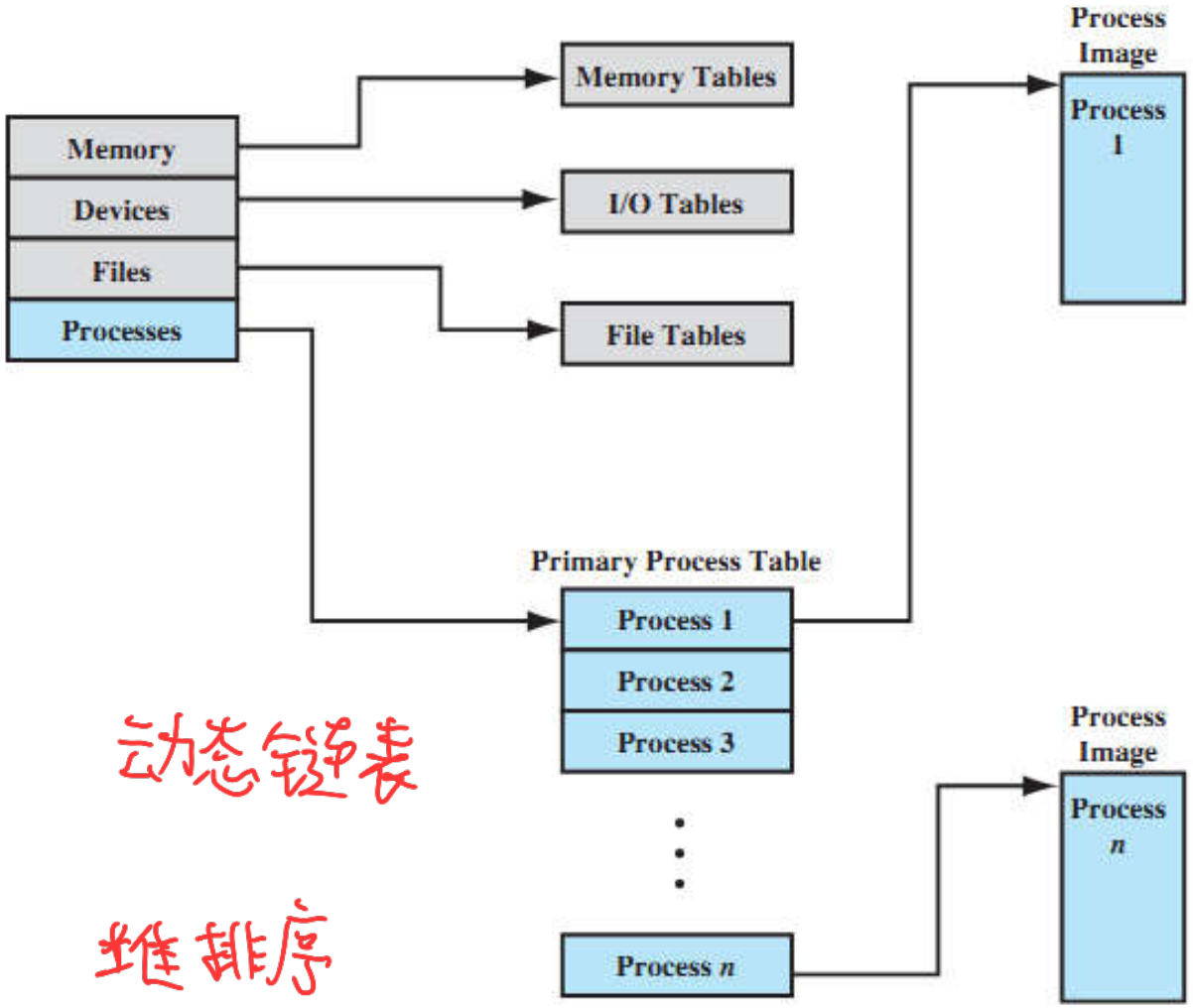
- Typical Process Table Implementation.



running process

Basic Concepts

- General Structure of OS Control Tables.



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

- *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

- 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 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.

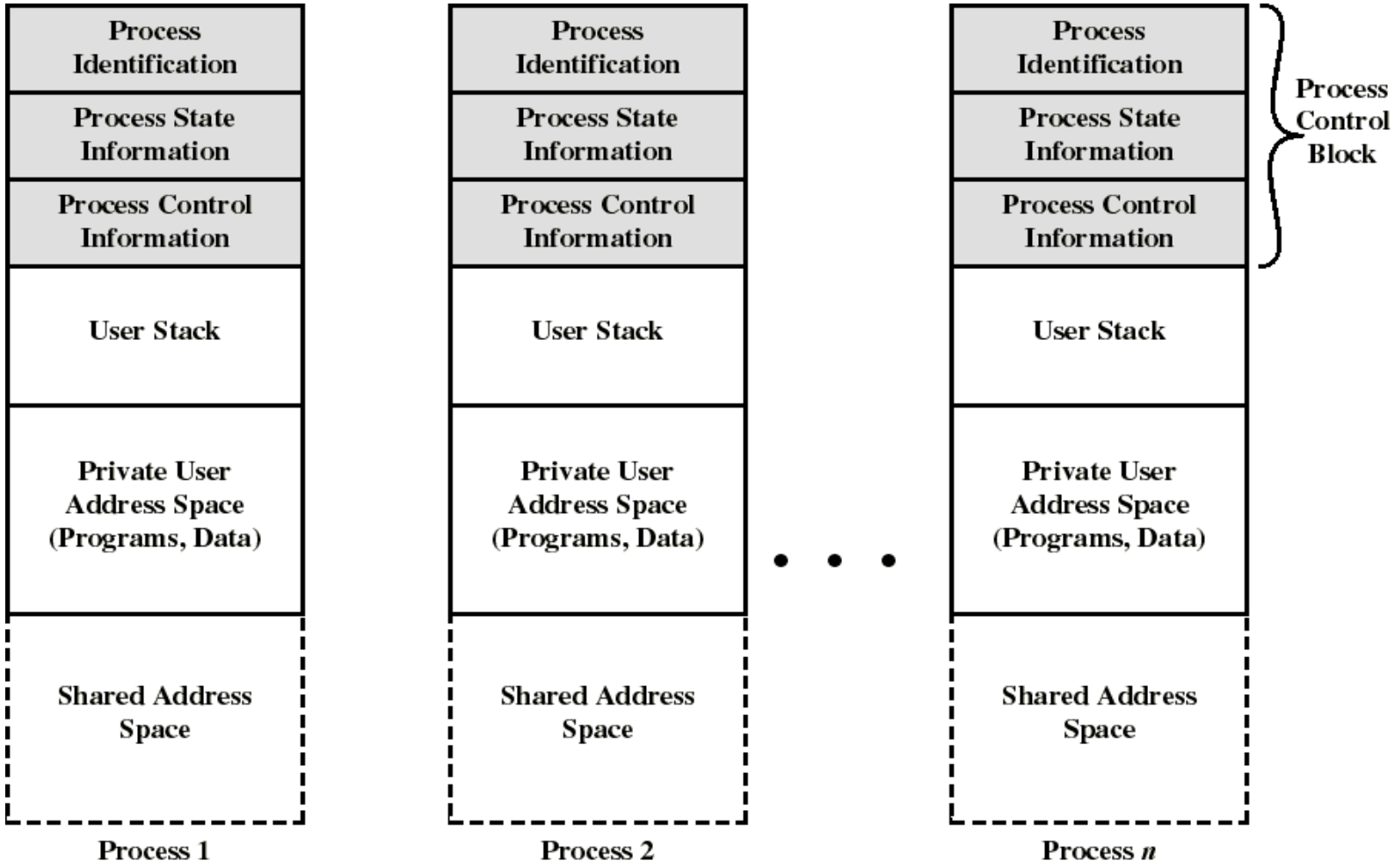
Process control block (PCB)

Process 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 Control Block (PCB)

- Process Location Information.



Process Images in Memory

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 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 Control Block (PCB)

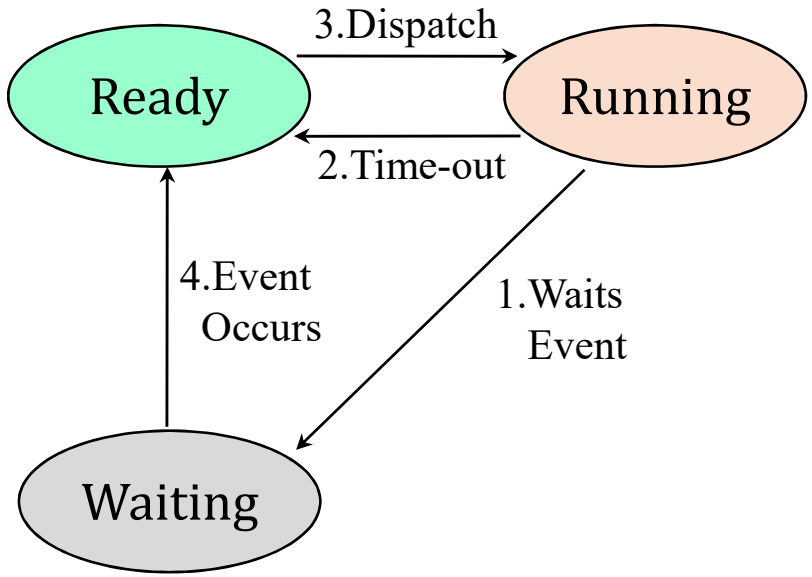
- Process Control Information (cont.)
  - Program counter
    - indicates the address of the next instruction to be executed for this process.
  - Accounting information
    - includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
  - I/O status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

- Three-state Process Model.



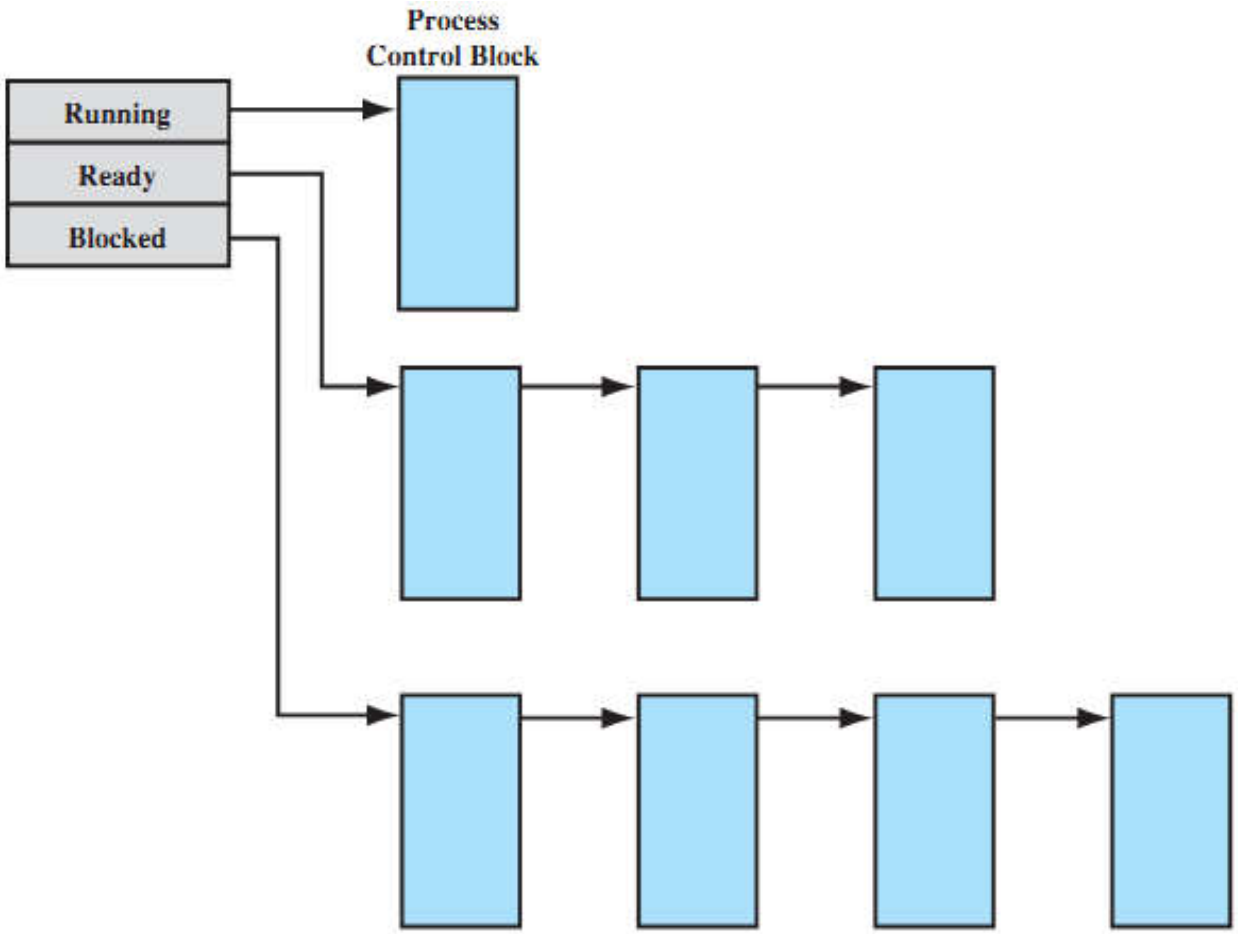
1. Process blocks for input.
2. Scheduler picks another process.
3. Scheduler picks this process.
4. Input becomes available.

Process States and Transitions

- Three-state Process Model
  - Process State Transitions
    - Ready → Running
      - When it is time, the dispatcher selects a new process to run.
    - Running → 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 → 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.
        - initiates I/O and must wait for the result.
        - waiting for a process to provide input.
    - Waiting → Ready
      - When the event for which the process was waiting occurs.

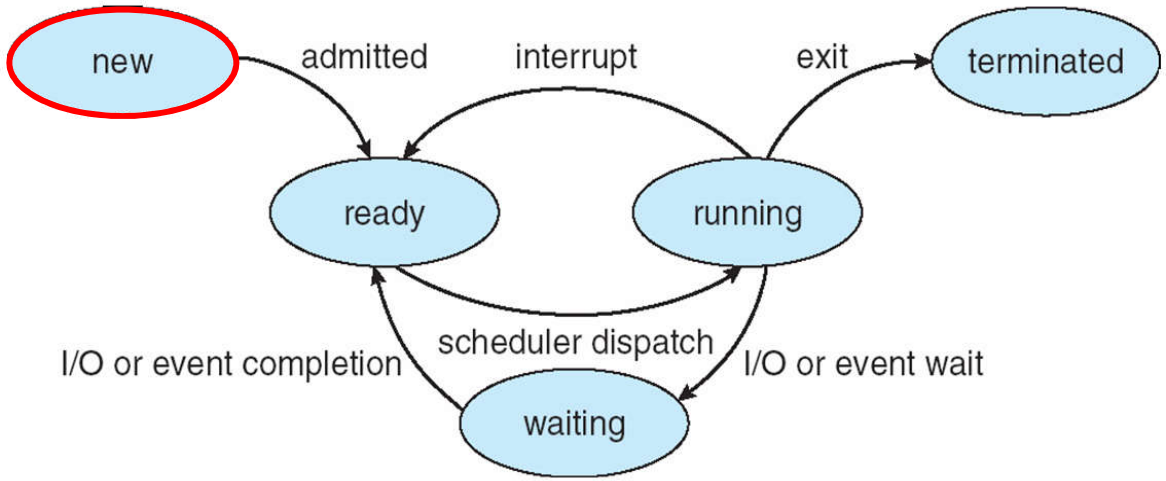
Process States and Transitions

- Three-state Process Model
  - Process List Structures.



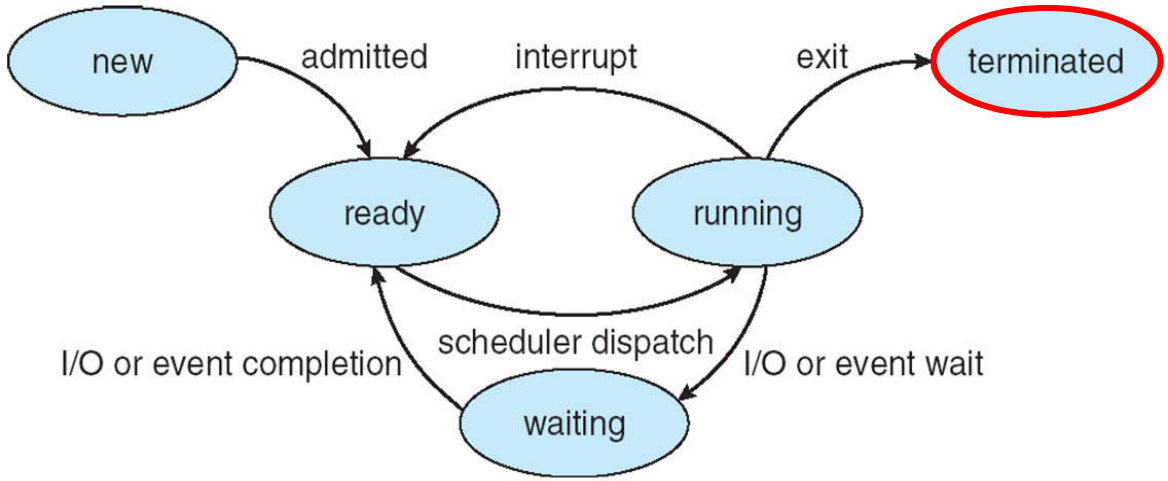
Process States and Transitions

- 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.



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.

Process Creation

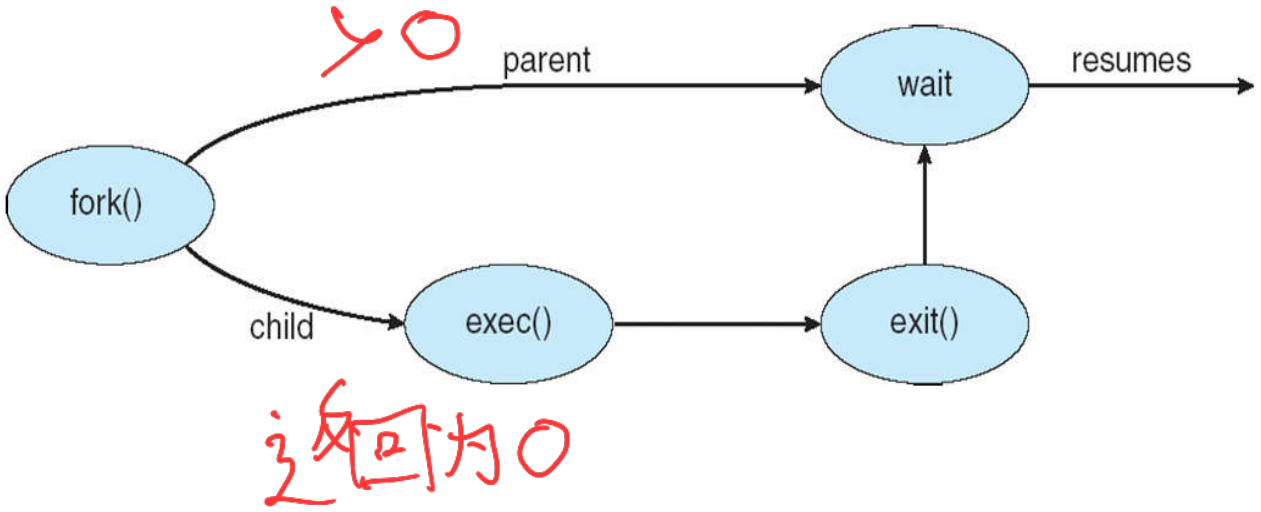
- Details in Process Creating
  - Parent process create children processes, which, in turn create other processes, forming a *tree* of processes.
  - Possible resource sharing:
    - Parent process and children processes share all resources.
    - Children processes share subset of parent's resources.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

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.

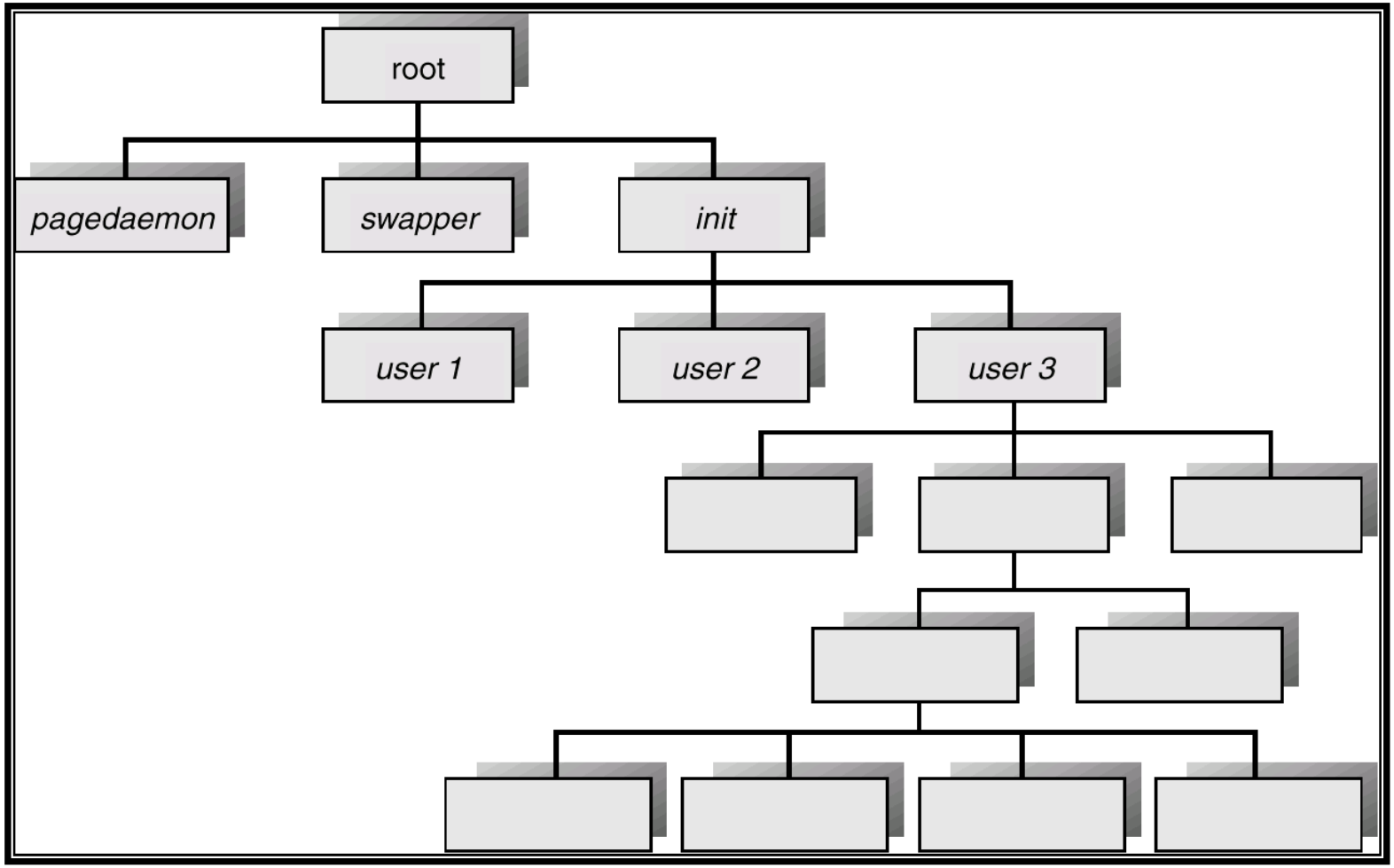
Process Creation

- 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.



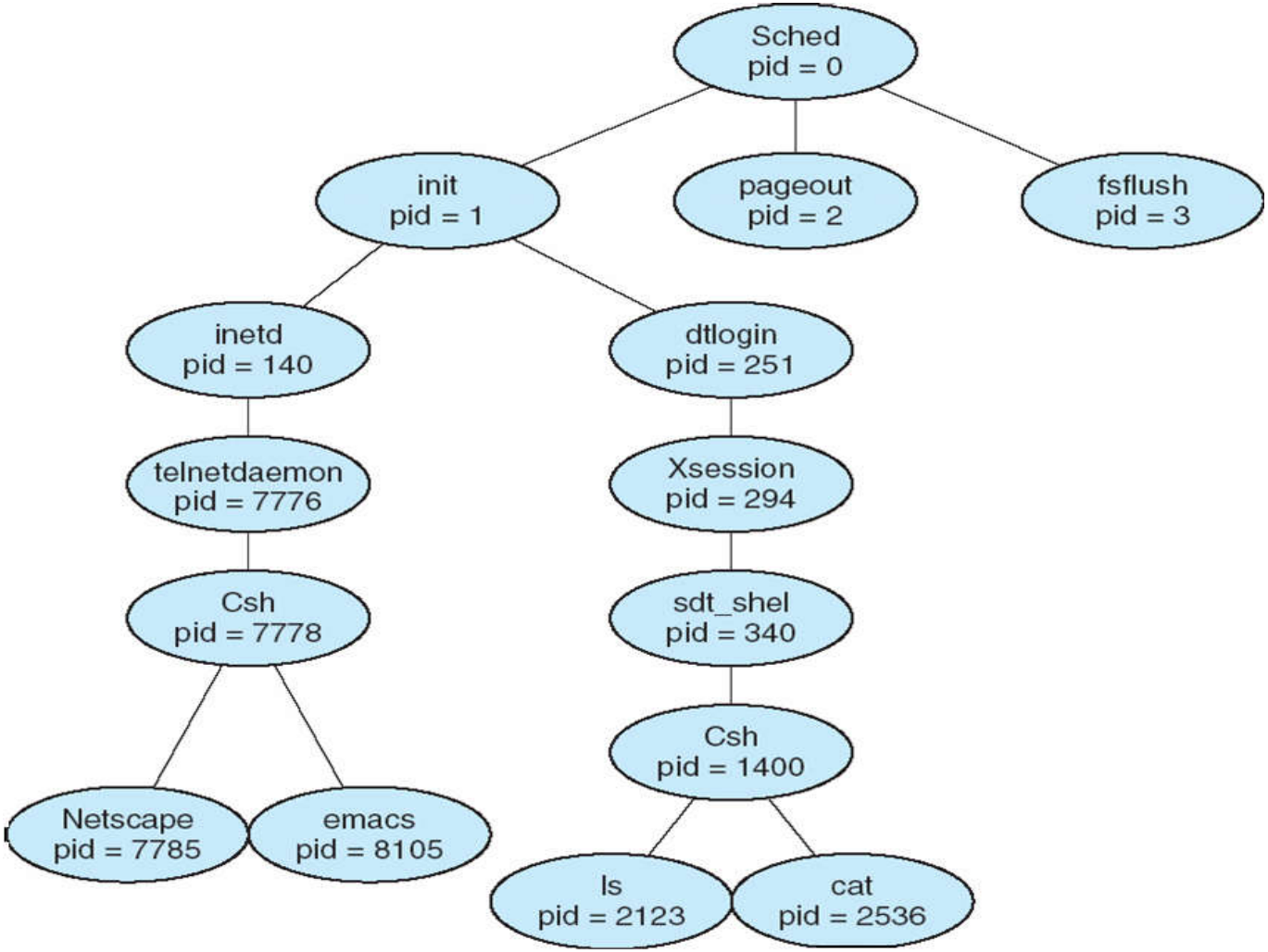
Process Creation

- A Tree of Processes on UNIX.



Process Creation

- A Tree of Processes on Typical Solaris.



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 error: ");
        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;
}
```

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$
```

子
父

虚拟地址一样

物理地址不一样

```
            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;
}
```

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.

```
            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;
}
```

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;
}
```

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking 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 = vfork(); /* child shares parent's address space */
    if (childpid < 0) {
        perror("fork error: ");
        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;
}
```

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking 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;
```

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Child pro pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Child taking a nap ...
Child waking up!
Parent pro pid = 26377, child pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Testing point by 26377
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-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;
}
```

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking 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;
```

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Child pro pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Child taking a nap ...
Child waking up!
Parent pro pid = 26377, child pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Testing point by 26377
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-t
```

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 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;
```

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Child pro pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Child taking a nap ...
Child waking up!
Parent pro pid = 26377, child pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Testing point by 26377
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-
```

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;
}
```

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking 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;
```

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Child pro pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Child taking a nap ...
Child waking up!
Parent pro pid = 26377, child pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Testing point by 26377
isscgy@ub
```

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;
}
```

Process Creation

- Algorithm 6-2: vfork-demo.c (vforking 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;
```

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-2-vfork-demo.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Child pro pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Child taking a nap ...
Child waking up!
Parent pro pid = 26377, child pid = 26378, count = 2 (addr = 0x7ffce0f96440)
Testing point by 26377
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-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;
}
```

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.

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.

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.

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.

Process Termination

- Algorithm 6-3: fork-demo-nowait.c (fork, execv 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 error: ");
        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;
}
```

no wait

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-3-fork-demo-nowait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Parent pro pid = 26439, child pid = 26440, count = 1 (addr = 0x7ffd221e7ca0)

Testing point by 26439
child pro pid = 26440, count = 2 (addr = 0x7ffd221e7ca0)
child sleeping ...
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:01 bash
1 S  1000  26440   1422  0  80   0 -  1128 hrtime pts/0    00:00:00 a.out
0 R  1000  26441  16125  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$
child waking up!

Testing point by 26440
```

```
            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;
}
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-3-fork-demo-nowait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Parent pro pid = 26439, child pid = 26440, count = 1 (addr = 0x7ffd221e7ca0)

Testing point by 26439
child pro pid = 26440, count = 2 (addr = 0x7ffd221e7ca0)
child sleeping ...
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:01 bash
1 S  1000  26440   1422  0  80   0 -  1128 hrtime pts/0    00:00:00 a.out
0 R  1000  26441  16125  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt
child waking up!

Testing point by 26440
```

The parent process terminated with an orphan of pid = 26440 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;
}
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-3-fork-demo-nowait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
Parent pro pid = 26439, child pid = 26440, count = 1 (addr = 0x7ffd221e7ca0)

Testing point by 26439
child pro pid = 26440, count = 2 (addr = 0x7ffd221e7ca0)
child sleeping ...
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:01 bash
1 S  1000  26440   1422  0  80   0 -  1128 hrtime pts/0    00:00:00 a.out
0 R  1000  26441  16125  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$
child waking up!

Testing point by 26440
```

What happens here? The control is not automatically back to the BASH.

```
id = %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 error: ");
        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;
}
```

等待儿子结束

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-4-fork-demo-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
child pro pid = 26620, count = 2 (addr = 0x7ffdca18e4cc)
child sleeping ...

child waking up!

Testing point by 26620
Parent pro pid = 26619, terminated pid = 26620, count = 1 (addr = 0x7ffdca18e4cc)

Testing point by 26619
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps
   PID TTY          TIME CMD
 16125 pts/0    00:00:01 bash
 26621 pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-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/VM-Shared/OS-test-2020$ gcc alg.6-4-fork-demo-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
child pro pid = 26620, count = 2 (addr = 0x7ffdca18e4cc)
child sleeping ...

child waking up!

Testing point by 26620
Parent pro pid = 26619, terminated pid = 26620, count = 1 (addr = 0x7ffdca18e4cc)

Testing point by 2661
isscgy@ubuntu:/mnt/hg
   PID TTY          T
 16125 pts/0    00:00:01 bash
 26621 pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-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;
}
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-4-fork-demo-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
child pro pid = 26620, count = 2 (addr = 0x7ffdca18e4cc)
child sleeping ...

child waking up!

Testing point by 26620
Parent pro pid = 26619, terminated pid = 26620, count = 1 (addr = 0x7ffdca18e4cc)

Testing point by 26619
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps
   PID TTY          T
 16125 pts/0    00:00
 26621 pts/0    00:00
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-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;
}
```

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 = atoi(argv[1]);

    if ( secnd < 0 || secnd > 10) {
        printf("Sleeping time = %d out of range (0-10)!\n", secnd);
        return 0;
    }

    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).

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

int main(void)
{
    pid_t childpid;

    childpid = vfork(); /* child shares parent's address space */
    if (childpid < 0) {
        perror("fork error: ");
        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 *argv[] = {"./alg.6-5-0.sleeper.o", "5", NULL};
            printf("child execv() a sleeper: %s\n\n",argv[0]);
            execv("./alg.6-5-0-sleeper.o", argv); /* parent resume at the point 'execv'
called */
        }
        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);
            printf("\nwait() returns childpid = %d\n", retpid);
            perror("\nwait() message:");
            return EXIT_SUCCESS;
        }
}
```

儿子调用 execv 意味着自杀，建立一个新的进程，继承儿子pid与父亲进程同步

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out 20
This is child, pid = 26755, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0.sleeper.o

This is parent, pid = 26754, childpid = 26755

sleeper pid = 26755, ppid = 26754
sleeper is taking a nap for 5 seconds

sleeper wakes up and returns

wait() returns childpid = 26755

wait() message:: Success
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$
```

sleeper 仍是父亲的儿子
但不共享pvm

```
                printf("This is parent, pid = %d, childpid = %d \n",getpid(), childpid);
                    /* parent executed this statement during the EXECV time */
                int retpid = wait(0);
                printf("\nwait() returns childpid = %d\n", retpid);
                perror("\nwait() message:");
                return EXIT_SUCCESS;
            }
        }
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out 20
This is child, pid = 26755, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0.sleeper.o

This is parent, pid = 26754, childpid = 26755

sleeper pid = 26755, ppid = 26754
sleeper is taking a nap for 5 seconds

sleeper wakes up and returns

wait() returns childpid = 26755

wait() message:: Success
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$
```

The sleeper inherits the pid (16890) of the vforked child

```
                printf("This is parent, pid = %d, childpid = %d \n",getpid(), childpid);
                    /* parent executed this statement during the EXECV time */
                int retpid = wait(0);
                printf("\nwait() returns childpid = %d\n", retpid);
                perror("\nwait() message:");
                return EXIT_SUCCESS;
            }
        }
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out 20
This is child, pid = 26755, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0.sleeper.o

This is parent, pid = 26754, childpid = 26755

sleeper pid = 26755, ppid = 26754
sleeper is taking a nap for 5 seconds

sleeper wakes up and returns

wait() returns childpid = 26755

wait() message:: Success
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-t
```

parent pro resumed at the point 'execv' called. The 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.

```
                printf("This is parent, p
                    /* parent executed th
                int retpid = wait(0);
                printf("\nwait() returns childpid = %d\n", retpid);
                perror("\nwait() message:");
                return EXIT_SUCCESS;
            }
        }
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-5-vfork-execv-wait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out 20
This is child, pid = 26755, taking a nap for 2 seconds ...
child waking up and again execv() a sleeper: ./alg.6-5-0.sleeper.o

This is parent, pid = 26754, childpid = 26755

sleeper pid = 26755, ppid = 26754
sleeper is taking a nap for 5 seconds

sleeper wakes up and returns

wait() returns childpid = 26755

wait() message:
isscgy@ubuntu:/
```

without wait(), the spawned sleeper pro may become an orphan

```
= %d \n",getpid(), childpid);
ng the EXECV time */
        int retpid = wait(0);
        printf("\nwait() returns childpid = %d\n", retpid);
        perror("\nwait() message:");
        return EXIT_SUCCESS;
    }
}
```

Process Termination

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

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

int main(void)
{
    pid_t childpid;

    childpid = vfork(); /* child shares parent's address space */
    if (childpid < 0) {
        perror("fork error: ");
        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 *argv[] = {"./alg.6-5-0.sleeper.o", "5", NULL};
            printf("child execv() the sleeper: %s %s\n\n",argv[0], argv[1]);
            execv("./alg.6-5-0-sleeper.o", argv);
        }
        else { /* This is parent pro, start when the vforked child terminated */
            printf("This is parent, pid = %d, childpid = %d \n",getpid(), childpid); /*
parent excuted 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 */
    }
}
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
This is child, pid = 26835, taking a nap for 2 seconds ...
child execv() the sleeper: ./alg.6-5-0.sleeper.o 5

This is parent, pid = 26834, childpid = 26835
parent calling shell ps

sleeper pid = 26835, ppid = 26834
sleeper is taking a nap for 5 seconds
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:02 bash
0 S  1000  26834  16125  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000  26835  26834  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000  26836  26834  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000  26837  26836  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:02 bash
0 S  1000  26835   1422  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000  26838  16125  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$
sleeper wakes up and returns
```

Process Termination

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

```
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ gcc alg.6-6-vfork-execv-nowait.c
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ./a.out
This is child, pid = 26835, taking a nap for 2 seconds ...
child execv() the sleeper: ./alg.6-5-0.sleeper.o 5

This is parent, pid = 26834, childpid = 26835
parent calling shell ps

sleeper pid = 26835, ppid = 26834
sleeper is taking a nap for 5 seconds
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:02 bash
0 S  1000  26834  16125  0  80   0 -  1128 wait   pts/0    00:00:00 a.out
0 S  1000  26835  26834  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000  26836  26834  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000  26837  26836  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  16125   1890  0  80   0 -  6157 wait   pts/0    00:00:02 bash
0 S  1000  26835   1422  0  80   0 -  1128 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000  26838  16125  0  80   0 -  7667 -      pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/hgfs/VM-Shared/OS-test-2020$
sleeper wakes up and returns
```

parent pro exited before sleeper terminated, and sleeper pro became an orphan