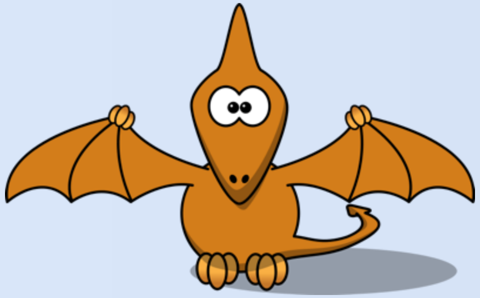
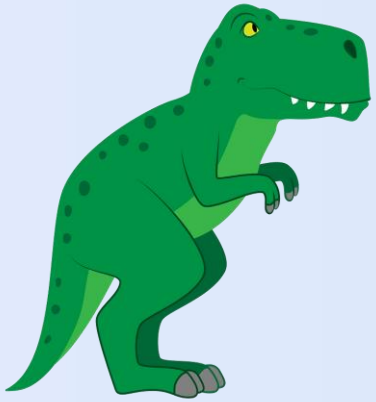
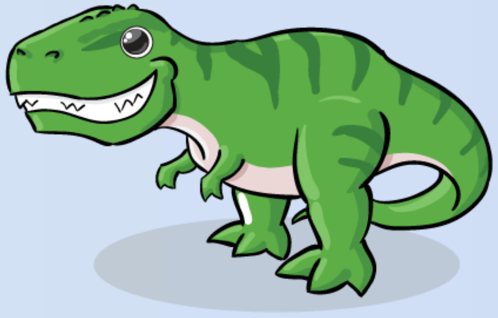


Chapter 3.

Processes 1.



Operating System Concepts (10th Ed.)



3.1 Process Concept

- A *process* is a program in execution.

  - A process is the unit of work in an operating system.

  - A process will need certain resources to accomplish its task.
    - CPU time,
    - memory,
    - files,
    - and I/O devices.

3.1 Process Concept

- The memory layout of a process is divided into multiple sections:
  - Text section:
    - the executable code
  - Data section:
    - global variables
  - Heap section:
    - memory that is dynamically allocated during program run time
  - Stack section:
    - temporary data storage when invoking functions
    - such as function parameters, return addresses, and local variables

3.1 Process Concept

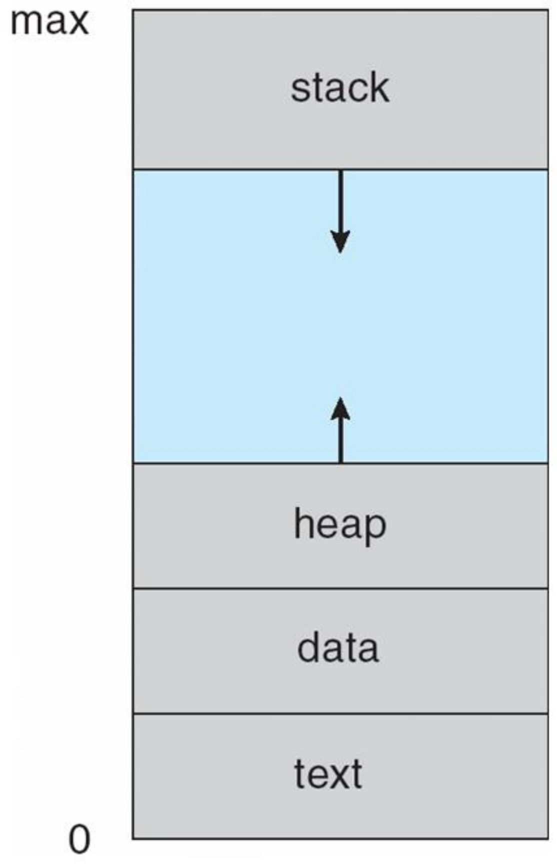
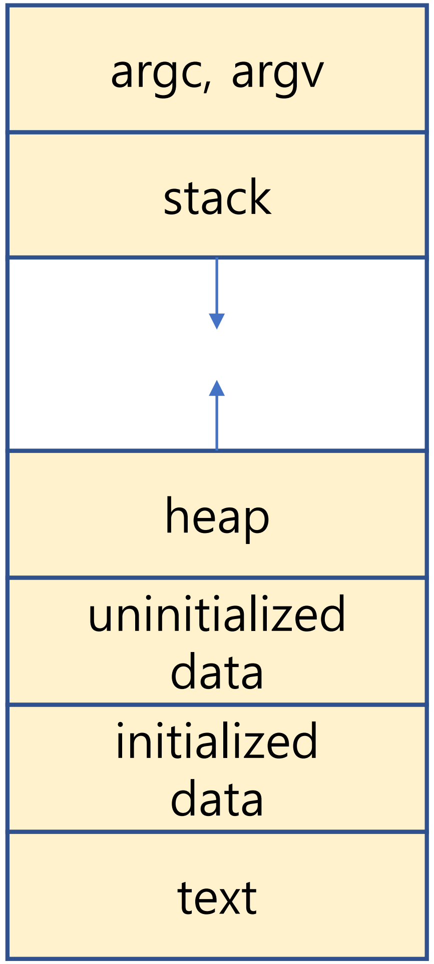


Figure 3.1 *Layout of a process in memory.*

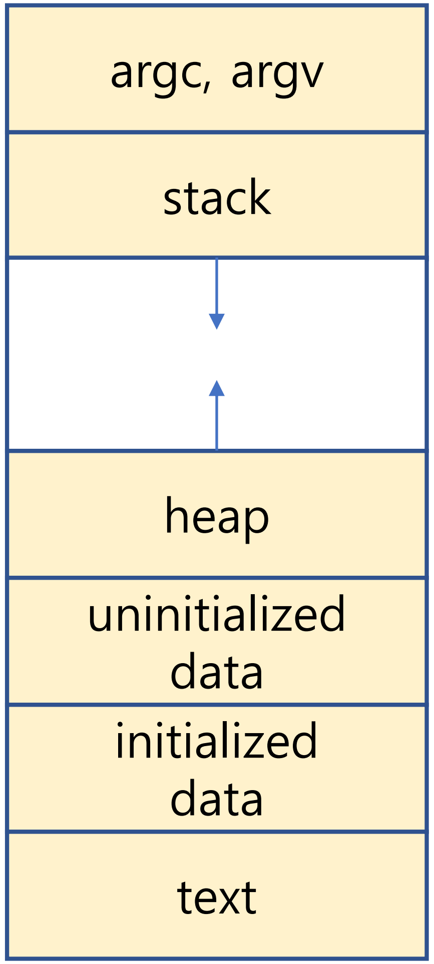
3.1 Process Concept

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for (i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```



3.1 Process Concept

```
$ gcc 3.1_memory_layout.c
$ size ./a.out
   text    data     bss     dec     hex filename
   1603     604      12    2219     8ab ./a.out
```



3.1 Process Concept

- As a process executes, it changes its state.
  - New: the process is being created.
  - Running: Instructions are being executed.
  - Waiting: the process is waiting for some event to occur.
    - such as an I/O completion or reception of a signal.
  - Ready: the process is waiting to be assigned to a processor.
  - Terminated: the process has finished execution.

3.1 Process Concept

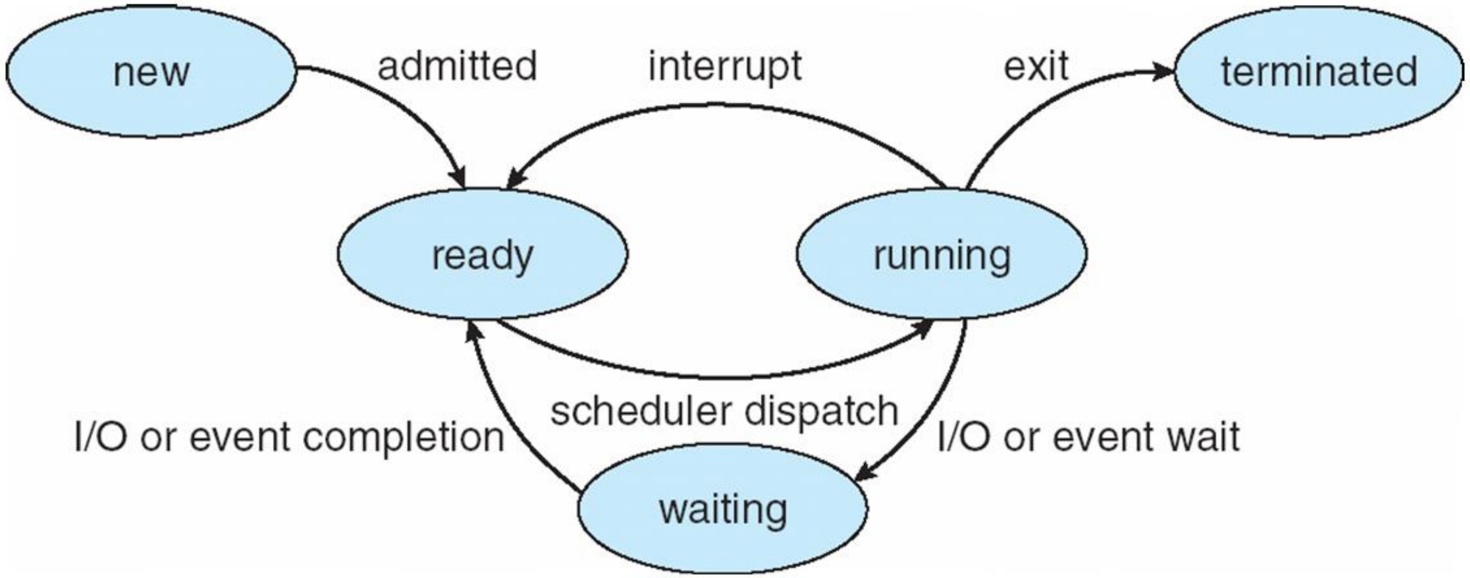


Figure 3.2 *Diagram of process state.*

3.1 Process Concept

- PCB (Process Control Block) or TCB (Task Control Block)
  - Each process is represented in the operating system by the PCB.
- A PCB contains many pieces of information associated with a specific process:
  - Process state
  - Program counter
  - CPU registers
  - CPU-scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

3.1 Process Concept

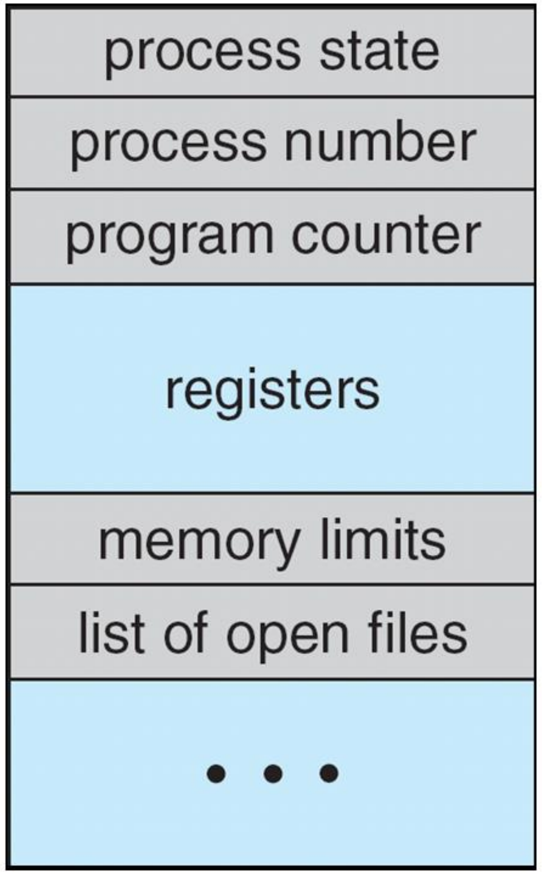


Figure 3.3 *Process control block (PCB).*

3.1 Process Concept

- A process is
  - a program that performs a ***single thread of execution***.
  - The single thread of control allows the process to perform
    - only one task at a time.
  - Modern operating systems have extended the process concept
    - to allow a process to have multiple threads of execution
    - and thus to perform more than one task at a time.

- A thread is a *lightweight* process.
  - Chapter 4 explores multithreading in detail.

3.2 Process Scheduling

- The objective of multiprogramming is
  - to have some process running at all times
  - so as to maximize CPU utilization.
- The objective of time sharing is
  - to switch a CPU core among processes so frequently
  - that users can interact with each program while it is running.

3.2 Process Scheduling

- Scheduling Queues:
  - As processes enter the system, they are put into a ready queue,
    - where they are ready and waiting to execute on a CPU's core.
  - Processes that are waiting for a certain event to occur
    - are placed in a wait queue.
  - These queues are generally implemented
    - in the linked lists of PCBs.

3.2 Process Scheduling

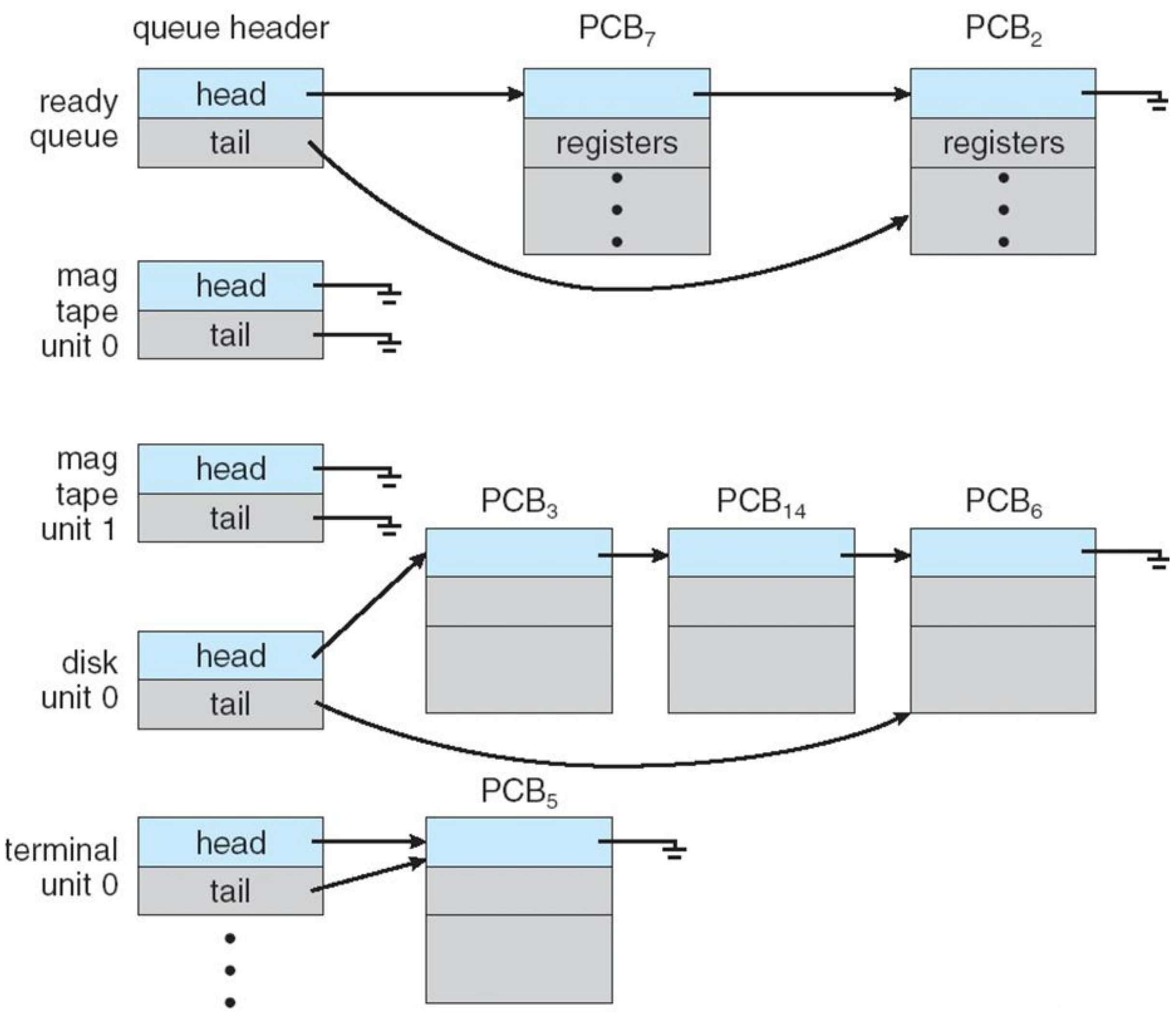


Figure 3.4 *The ready queue and wait queues.*

3.2 Process Scheduling

- Queueing Diagram
  - as a common representation of process scheduling.

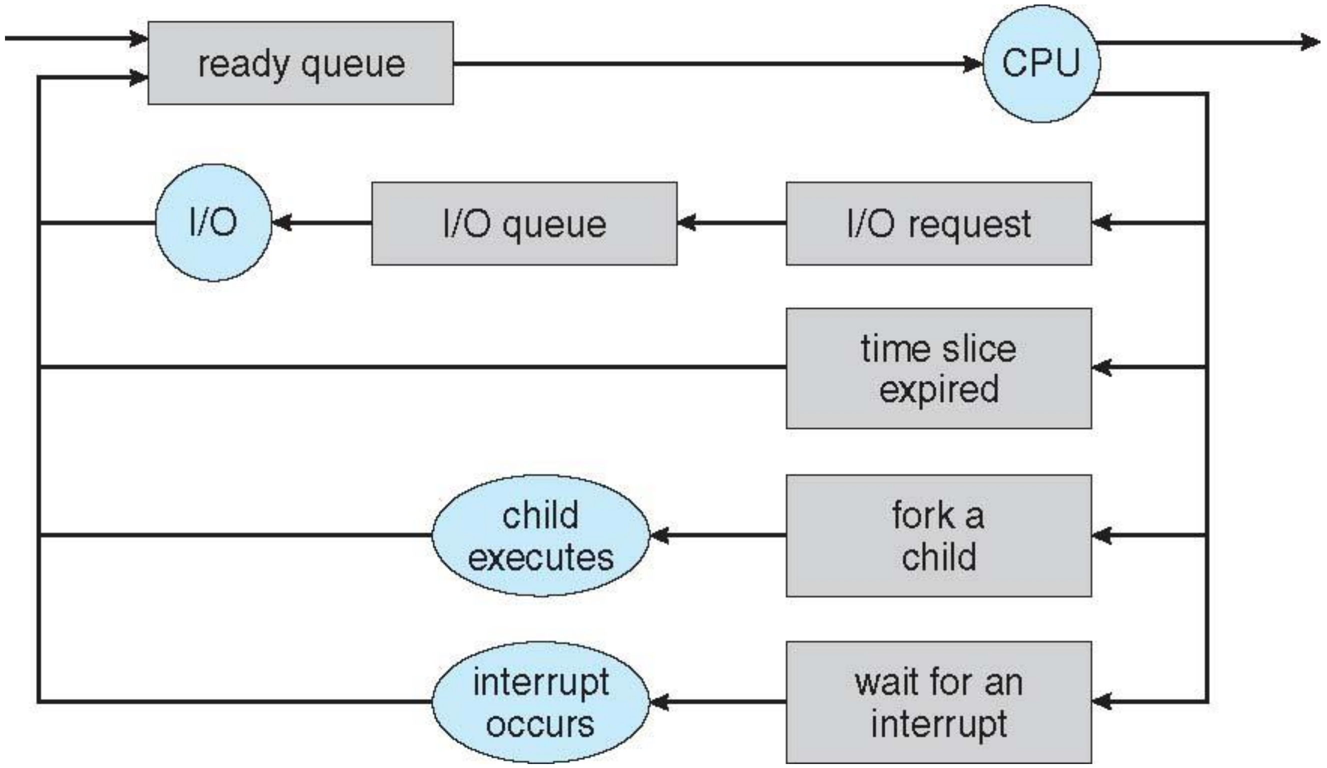


Figure 3.5 *Queueing-diagram representation of process scheduling.*

3.2 Process Scheduling

- Context Switch
  - The context of a process is represented in the PCB.
  - When an interrupt occurs,
    - the system ***saves*** the current ***context*** of the running process,
    - so that, later, it can ***restore*** that ***context*** when it should be resumed.
  - The context switch is a task that
    - switches the CPU core to another process.
    - performs a *state save* of the current process
    - and a *state restore* of a different process.

3.2 Process Scheduling

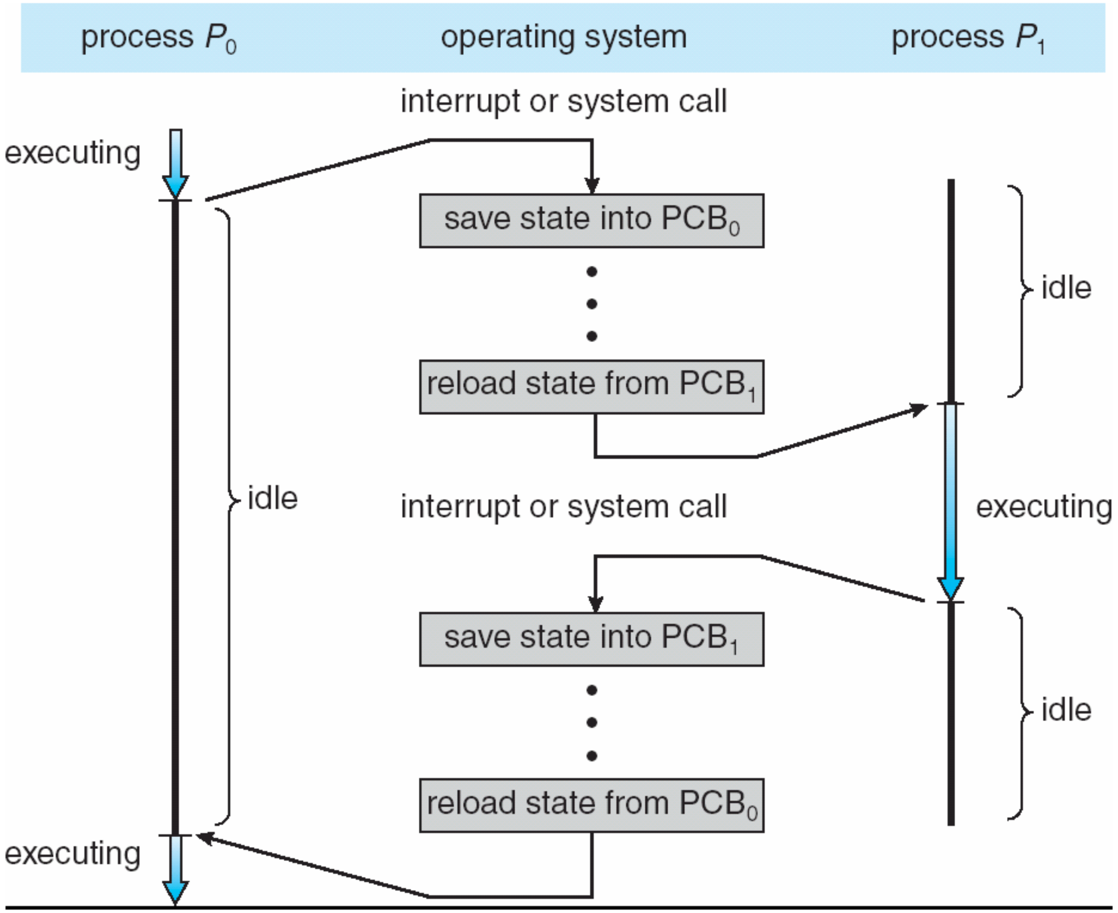


Figure 3.6 *Diagram showing context switch from process to process.*

3.3 Operations on Processes

- An operating system must provide a mechanism for
  - process creation,
  - and process termination.

- A process may create several new processes
  - the creating process: a *parent* process.
  - a newly created process: a *child* process.

3.3 Operations on Processes

- A *tree* of processes

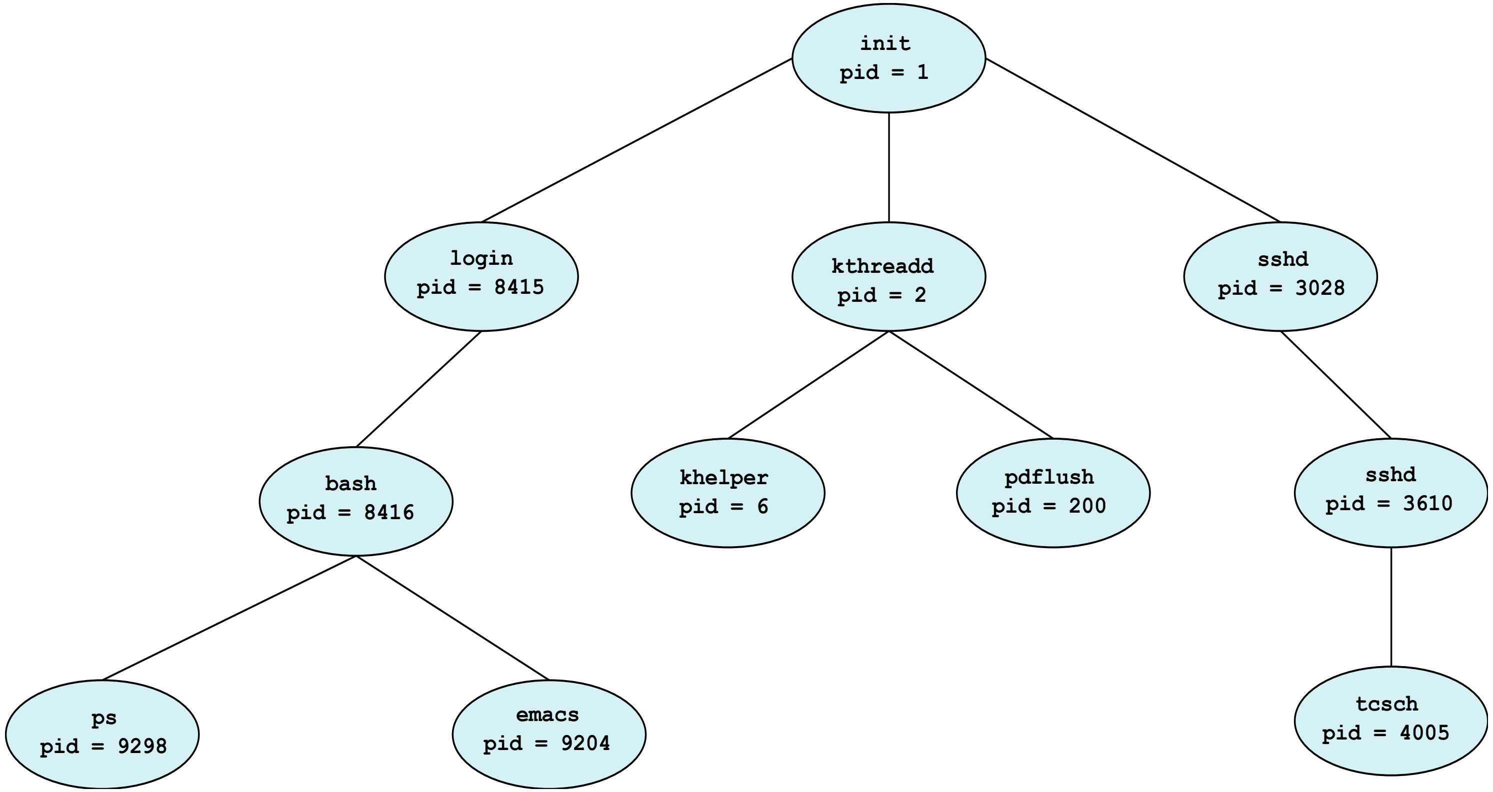


Figure 3.7 A tree of processes on a typical Linux system.

3.3 Operations on Processes

- Two possibilities for execution
  - The parent continues to ***execute concurrently*** with its children.
  - The parent ***waits*** until some or all of its children have terminated.

- Two possibilities of address-space
  - The child process is a ***duplicate*** of the parent process.
  - The child process has a ***new program*** loaded into it.

3.3 Operations on Processes

```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main()
{
    pid_t pid;
    // fork a child process
    pid = fork();
    if (pid < 0) { // error occurred
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { // child process
        execlp("/bin/ls", "ls", NULL);
    }
    else { // parent process
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Figure 3.8 Creating a separate process using the UNIX fork() system call.

3.3 Operations on Processes

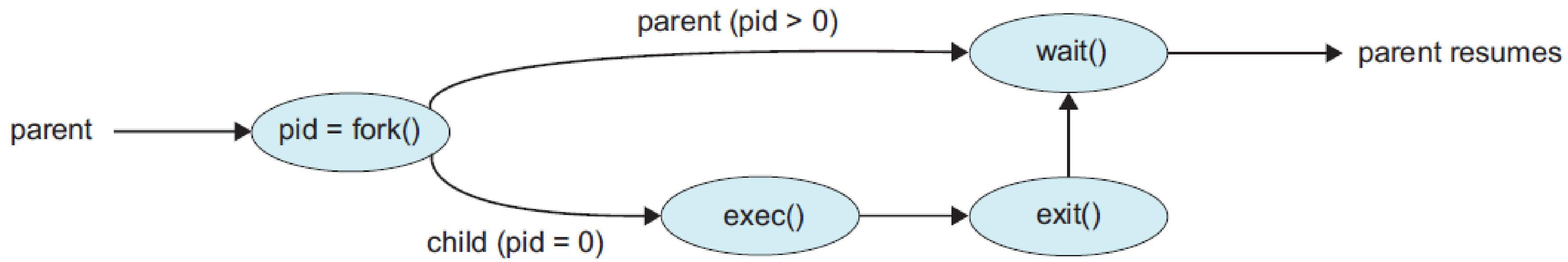


Figure 3.9 Process creation using the fork() system call.

3.3 Operations on Processes

- A process *terminates*
  - when it finishes executing its final statement
  - `exit()` system call: asks OS to delete it.
  - OS deallocates and reclaims all the resources:
    - allocated memories, open files, and I/O buffers, etc.

3.3 Operations on Processes

- Zombie and Orphan
  - *zombie* process: a process that has terminated,
    - but whose parent has not yet called wait().
  - *orphan* process: a process that has a parent process
    - who did not invoke wait() and instead terminated.

3.3 Operations on Processes

- In UNIX-like O/S,
  - A new process is created by the `fork()` system call.
  - The *child* process consists of
    - a ***copy of the address space*** of the *parent* process.
  - Both processes continue execution
    - at the instruction after the `fork()` system call.
  - With one difference:
    - the return code for the `fork()` is ***zero*** for the child process, whereas
    - the ***nonzero*** **`pid`** of the child is returned to the parent process.

3.3 Operations on Processes

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = fork();
    printf("Hello, Process!\n");
}
```

3.3 Operations on Processes

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = fork();
    printf("Hello, Process! %d\n", pid);
}
```

3.3 Operations on Processes

- After a `fork()` system call,
  - the parent can ***continue its execution***; or
  - if it has nothing else to do while the child runs,
    - it can issue a `wait()` system call
    - to move itself off the ready queue until the termination of the child.

3.3 Operations on Processes

```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main()
{
    pid_t pid;
    pid = fork();
    if (pid > 0)
        wait(NULL);
    printf("Hello, Process! %d\n", pid);
}
```

3.3 Operations on Processes

- Exercise 3.1 (p. 154)

```
int value = 5;

int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0) { // child process
        value += 15;
        return 0;
    }
    else if (pid > 0) { // parent process
        wait(NULL);
        printf("Parent: value = %d\n", value); // LINE A
    }
}
```

Figure 3.30 *What output will be at Line A?*

3.3 Operations on Processes

- Exercise 3.2 (p. 154)

```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

/*
 * How many processes are created?
 */
int main()
{
    fork(); // fork a child process
    fork(); // fork another child process
    fork(); // and fork another

    return 0;
}
```

Figure 3.31 *How many processes are created?*

3.3 Operations on Processes

- Exercise 3.11 (p. 905)

```
#include <stdio.h>
#include <unistd.h>

/*
 * How many processes are created?
 */
int main()
{
    int i;

    for (i = 0; i < 4; i++)
        fork();

    return 0;
}
```

Figure 3.32 *How many processes are created?*

3.3 Operations on Processes

- Exercise 3.12 (p. 905)

```
int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0) { // child process
        execlp("/bin/ls", "ls", NULL);
        printf("LINE J\n");
    }
    else if (pid > 0) { // parent process
        wait(NULL);
        printf("Child Complete\n");
    }

    return 0;
}
```

Figure 3.33 *When will LINE J be reached?*

- Exercise 3.13 (p. 905)

```
int main()
{
    pid_t pid, pid1;
    pid = fork();
    if (pid == 0) { // child process
        pid1 = getpid();
        printf("child: pid = %d\n", pid);    // A
        printf("child: pid1 = %d\n", pid1); // B
    }
    else if (pid > 0) { // parent process
        pid1 = getpid();
        printf("child: pid = %d\n", pid);    // C
        printf("child: pid1 = %d\n", pid1); // D
        wait(NULL);
    }
    return 0;
}
```

Figure 3.34 *What are the pid values?*

- Exercise 3.16 (p. 905)

```c
#define SIZE 5
int nums[SIZE] = {0, 1, 2, 3, 4};

int main()
{
    pid_t pid;
    int i;
    pid = fork();

    return 0;
}
```

```c
if (pid == 0) { // child process
    for (i = 0; i < SIZE; i++) {
        nums[i] *= i;
        printf("CHILD: %d \n", nums[i]); // LINE X
    }
}
else if (pid > 0) { // parent process
    wait(NULL);
    for (i = 0; i < SIZE; i++) {
        printf("PARENT: %d \n", nums[i]); // LINE X
    }
}
```

Figure 3.35 *What output will be at Line X and Line Y?*

Any Questions?