Processes (Part I)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

What Is A Process?

Process

An instance of a program running.

Program vs. Process

- Program is a passive entity stored on disk (executable file).

Program vs. Process

- Program is a passive entity stored on disk (executable file).
- Process is an active entity.

Program vs. Process

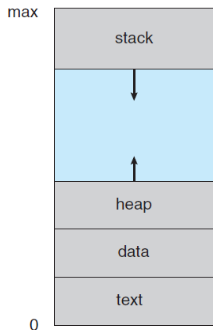
- Program is a passive entity stored on disk (executable file).
- Process is an active entity.
- Program becomes process when executable file loaded into memory.

Program vs. Process

- Program is a passive entity stored on disk (executable file).
- Process is an active entity.
- Program becomes process when executable file loaded into memory.
- One program can be several processes.

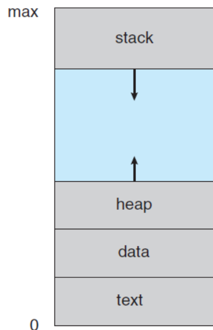
Parts of a Process

- A process is more than the program code.



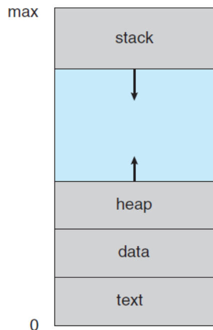
Parts of a Process

- A process is more than the program code.
- Multiple parts of a process:
  - The program code (text section).



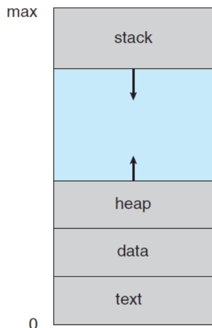
Parts of a Process

- A process is more than the program code.
- Multiple parts of a process:
  - The program code (text section).
  - Current activity, e.g., program counter, processor registers.



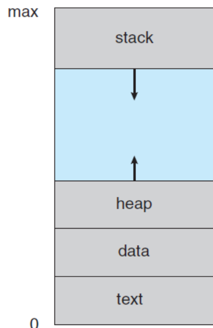
Parts of a Process

- A process is more than the program code.
- Multiple parts of a process:
  - The program code (text section).
  - Current activity, e.g., program counter, processor registers.
  - Data section containing global variables.



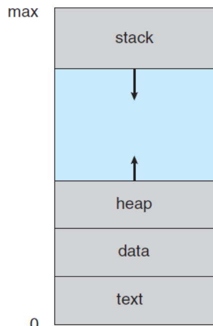
Parts of a Process

- A process is more than the program code.
- Multiple parts of a process:
  - The program code (text section).
  - Current activity, e.g., program counter, processor registers.
  - Data section containing global variables.
  - Stack containing temporary data.



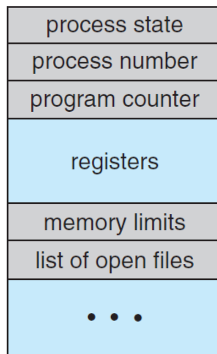
Parts of a Process

- A process is more than the program code.
- Multiple parts of a process:
  - The program code (text section).
  - Current activity, e.g., program counter, processor registers.
  - Data section containing global variables.
  - Stack containing temporary data.
  - Heap containing memory dynamically allocated during run time.



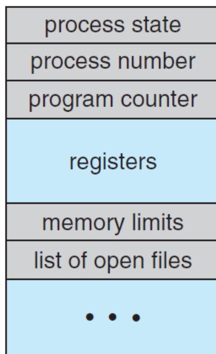
Process Control Block (PCB)

- The information of each process.



Process Control Block (PCB)

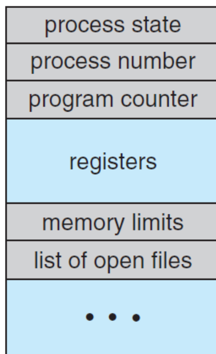
- The information of each process.



- Process state: running, waiting, etc

Process Control Block (PCB)

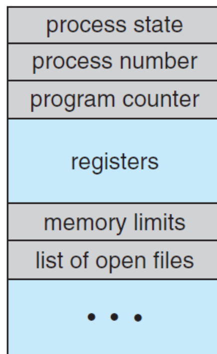
- The information of each process.



- Program counter: location of instruction to next execute

Process Control Block (PCB)

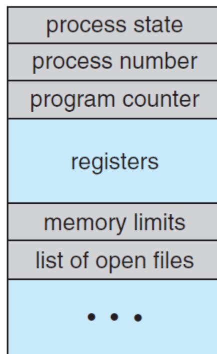
- The information of each process.



- CPU registers: contents of all process-centric registers

Process Control Block (PCB)

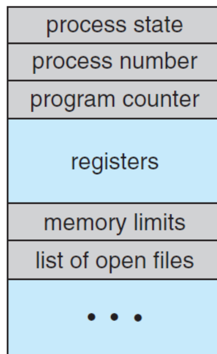
- The information of each process.



- CPU scheduling: priorities, scheduling queue pointers

Process Control Block (PCB)

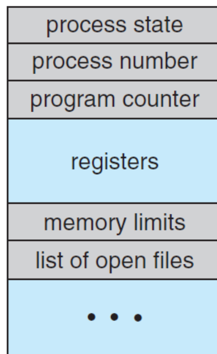
- The information of each process.



- Memory-management: memory allocated to the process

Process Control Block (PCB)

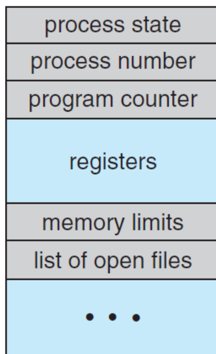
- The information of each process.



- Accounting: CPU used, clock time elapsed since start, time limits

Process Control Block (PCB)

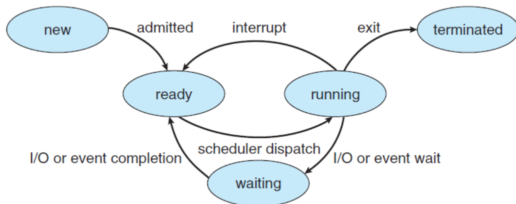
- The information of each process.



- I/O status: I/O devices allocated to process, list of open files

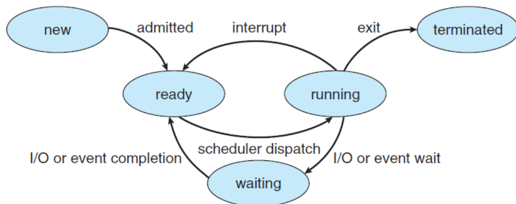
Process States

- As a process executes, it changes state.



Process States

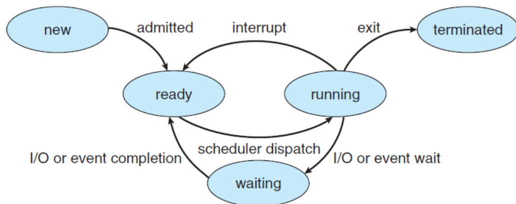
- As a process executes, it changes state.



- new: The process is being created.

Process States

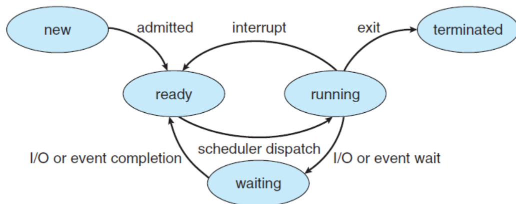
- As a process executes, it changes state.



- new: The process is being created.
- ready: The process is waiting to be assigned to a processor.

Process States

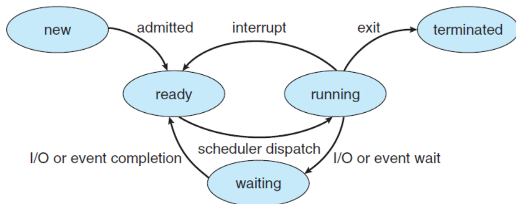
- As a process executes, it changes state.



- new: The process is being created.
- ready: The process is waiting to be assigned to a processor.
- running: Instructions are being executed.

Process States

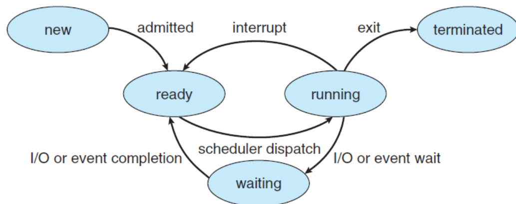
- As a process executes, it changes state.



- new: The process is being created.
- ready: The process is waiting to be assigned to a processor.
- running: Instructions are being executed.
- waiting: The process is waiting for some event to occur.

Process States

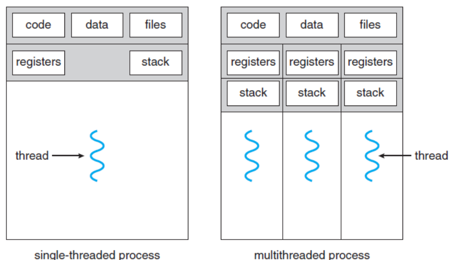
- As a process executes, it changes state.

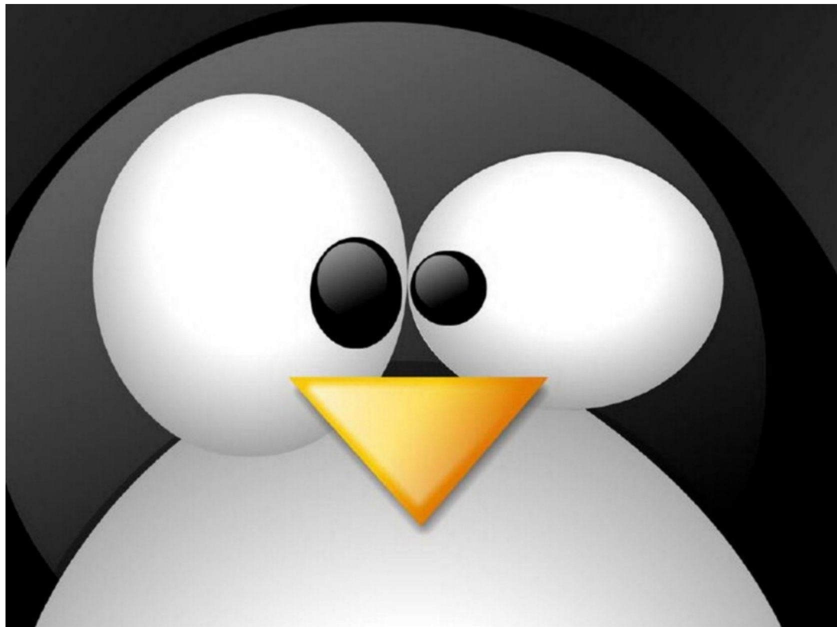


- new: The process is being created.
- ready: The process is waiting to be assigned to a processor.
- running: Instructions are being executed.
- waiting: The process is waiting for some event to occur.
- terminated: The process has finished execution.

Threads

- A process can have a single thread or multiple threads.
- Multi-thread process
  - Multiple program counters in a PCB: multiple locations can execute at once.





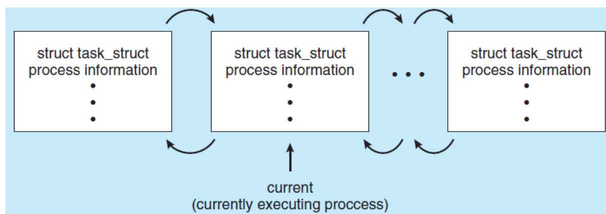
Process Data Structure in Linux Kernel (1/2)

- Represented by `task_struct` in the Linux kernel.
  - in the `<linux/sched.h>`

```
struct task_struct {
  volatile long state;
  long counter;
  struct task_struct *next_task, *prev_task;
  int pid;
  struct task_struct *p_pptr; // pointers to the parent
  struct task_struct *p_cptr; // pointers to the youngest child
  struct task_struct *p_ysptr // pointers to the younger sibling
  struct task_struct *p_osptr; // pointers to the older sibling
  struct wait_queue *wait_chldexit;
  unsigned short  uid, euid, suid, fsuid;
  ...
}
```

Process Data Structure in Linux Kernel (2/2)

- All active processes are represented using a doubly linked list of `task_struct`.




```
struct task_struct {
  ...
  struct task_struct *next_task, *prev_task;
  ...
}
```

Process ID (1/2)

- Each process is assigned a unique identifier, the process ID (PID).

Process ID (1/2)

- Each process is assigned a unique identifier, the process ID (PID).
- The kernel allocates PIDs to processes in a strictly linear fashion.

Process ID (1/2)

- Each process is assigned a unique identifier, the process ID (PID).
- The kernel allocates PIDs to processes in a strictly linear fashion.
- The maximum PID value of 32768.

```
cat /proc/sys/kernel/pid_max
```

Process ID (1/2)

- Each process is assigned a unique identifier, the process ID (PID).
- The kernel allocates PIDs to processes in a strictly linear fashion.
- The maximum PID value of 32768.

```
cat /proc/sys/kernel/pid_max
```

- The first process that the kernel executes after booting the system, is `init` process, with the PID 1.

Process ID (2/2)

- The PID is represented by the pid_t type.

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

pid_t getpid(void);
```

Process ID (2/2)

- The PID is represented by the `pid_t` type.
- The `getpid()` system call returns the PID of the invoking process.

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

pid_t getpid(void);
```

Process ID (2/2)

- The PID is represented by the pid_t type.
- The getpid() system call returns the PID of the invoking process.
- Defined in <sys/types.h>

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

pid_t getpid(void);
```

Process Scheduling

Process Scheduling

- The multiprogramming objective: to maximize CPU utilization.

Process Scheduling

- The multiprogramming objective: to maximize CPU utilization.

- The timesharing objective: to switch the CPU among processes so frequently that users can interact with each program while it is running.

Process Scheduling

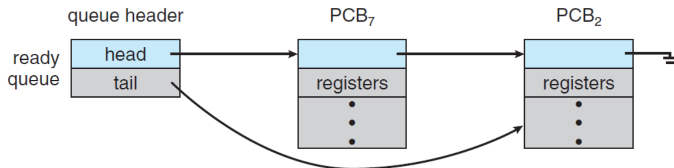
- The multiprogramming objective: to maximize CPU utilization.
- The timesharing objective: to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives: the process scheduler selects an available process for program execution on the CPU.
  - If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues (1/2)

- Job queue: set of all processes in the system.

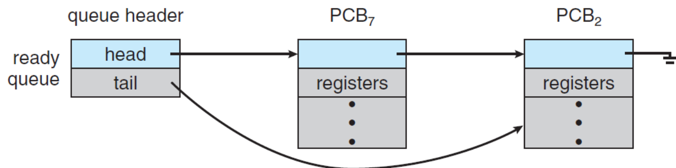
Scheduling Queues (1/2)

- Job queue: set of all processes in the system.
- Ready queue: set of all processes residing in main memory, ready and waiting to execute.



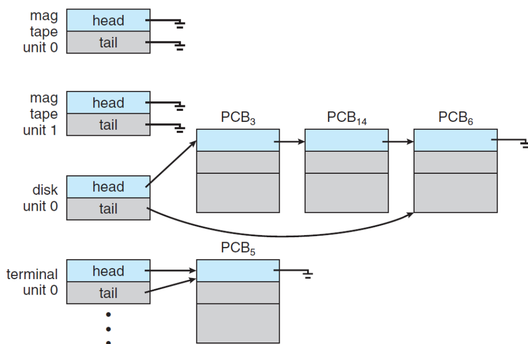
Scheduling Queues (1/2)

- Job queue: set of all processes in the system.
- Ready queue: set of all processes residing in main memory, ready and waiting to execute.
  - This queue is generally stored as a linked list.
  - Pointers to the first and final PCBs in the list.
  - Each PCB points to the next PCB in the ready queue.



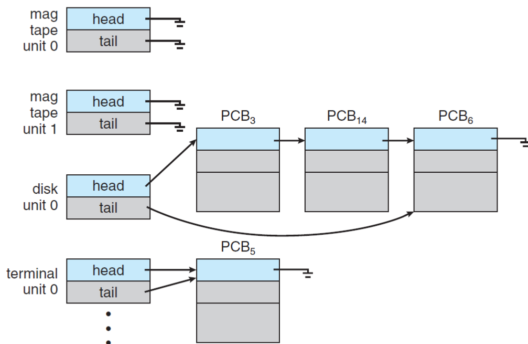
Scheduling Queues (2/2)

- Device queues: set of processes waiting for an I/O device.
  - Each device has its own device queue.



Scheduling Queues (2/2)

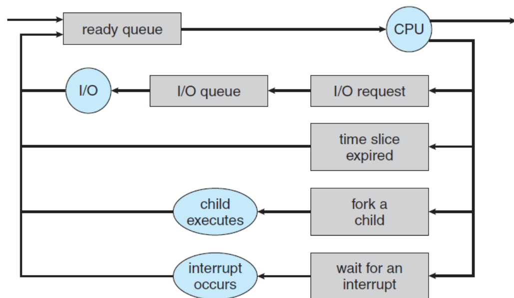
- Device queues: set of processes waiting for an I/O device.
  - Each device has its own device queue.



- Processes migrate among the various queues.

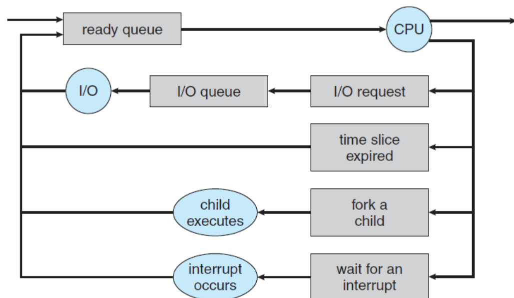
Queuing Diagram (1/2)

- A new process is initially put in the ready queue.
- It waits there until it is selected for execution or dispatched.



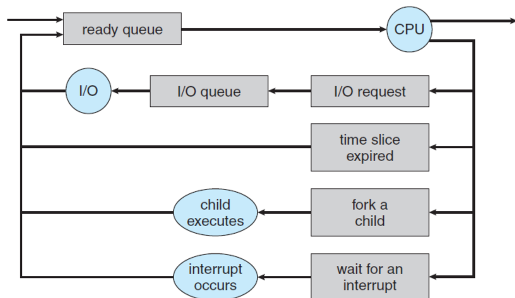
Queuing Diagram (2/2)

- After allocating the CPU to a process:



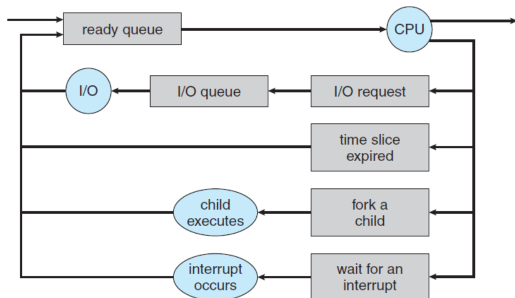
Queuing Diagram (2/2)

- After allocating the CPU to a process:
  - The process could issue an I/O request and be placed in an I/O queue.



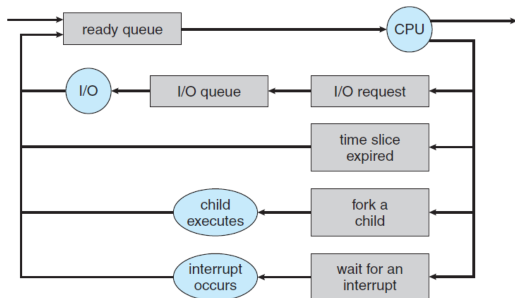
Queuing Diagram (2/2)

- After allocating the CPU to a process:
  - The process could issue an I/O request and be placed in an I/O queue.
  - The process could create a new child process and wait for the child's termination.



Queuing Diagram (2/2)

- After allocating the CPU to a process:
  - The process could issue an I/O request and be placed in an I/O queue.
  - The process could create a new child process and wait for the child's termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



Schedulers (1/3)

- Short-term scheduler (CPU scheduler)
  - Selects which process should be executed next and allocates CPU.
  - Invoked frequently (milliseconds): must be fast

Schedulers (2/3)

- Long-term scheduler (job scheduler)
  - Selects which processes should be brought into the ready queue.
  - Controls the degree of multiprogramming.
  - Invoked infrequently (seconds, minutes): may be slow.

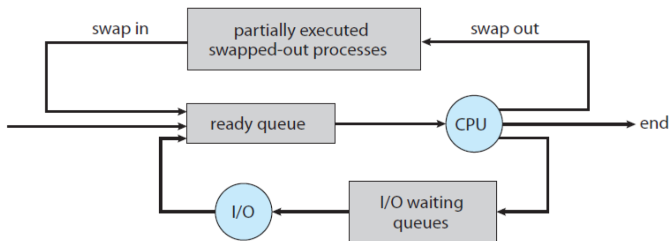
Schedulers (2/3)

- Long-term scheduler (job scheduler)
  - Selects which processes should be brought into the ready queue.
  - Controls the degree of multiprogramming.
  - Invoked infrequently (seconds, minutes): may be slow.

- It strives for good mix of I/O-bound and CPU-bound processes.
  - I/O-bound process: more time doing I/O than computations.
  - CPU-bound process: more time doing computations.

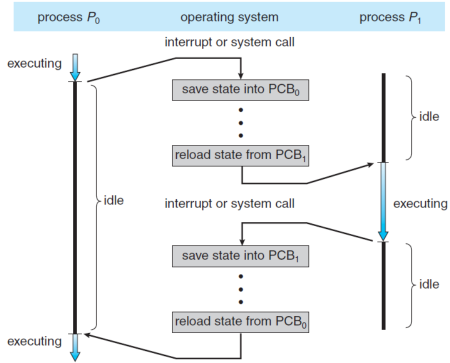
Schedulers (3/3)

- Medium-term scheduler
  - It can be added if degree of multiprogramming needs to decrease.
  - Remove process from memory, store on disk, bring back in from disk to continue execution: swapping



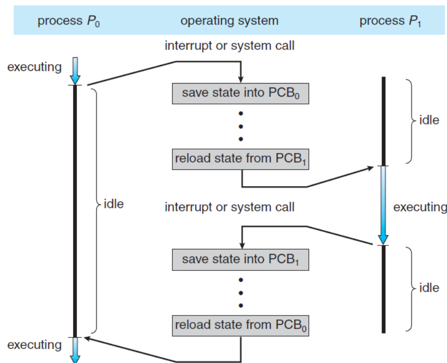
Context Switching (1/2)

- When CPU switches to another process:



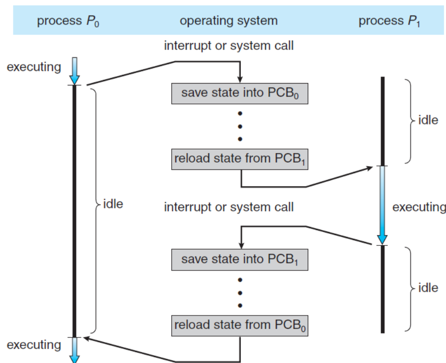
Context Switching (1/2)

- When CPU switches to another process:
  - The state of the old process is saved by the system.



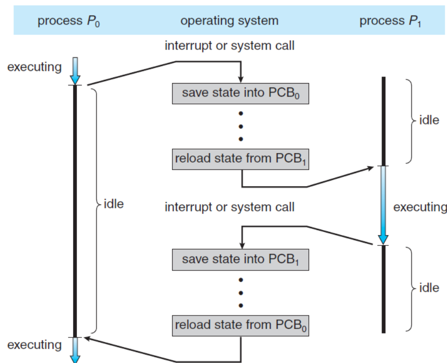
Context Switching (1/2)

- When CPU switches to another process:
  - The state of the old process is saved by the system.
  - The saved state of the new process is loaded via a context switch.



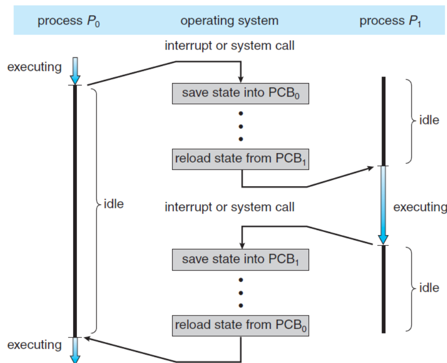
Context Switching (1/2)

- When CPU switches to another process:
  - The state of the old process is saved by the system.
  - The saved state of the new process is loaded via a context switch.
  - Called context switching.



Context Switching (1/2)

- When CPU switches to another process:
  - The state of the old process is saved by the system.
  - The saved state of the new process is loaded via a context switch.
  - Called context switching.



- Context of a process represented in the PCB.

Context Switching (2/2)

- Context-switch time is an overhead.
  - The system does no useful work while switching.
  - The more complex the OS and PCB $\rightarrow$ the longer the context switch.

Context Switching (2/2)

- Context-switch time is an overhead.
  - The system does no useful work while switching.
  - The more complex the OS and PCB $\rightarrow$ the longer the context switch.
- Time dependent on hardware support.
  - Some hardware provides multiple sets of registers per CPU $\rightarrow$ multiple contexts loaded at once.

Operations on Processes

Operations on Processes

- OS must provide mechanisms for:
  - Process creation
  - Process termination
  - ...

Process Creation

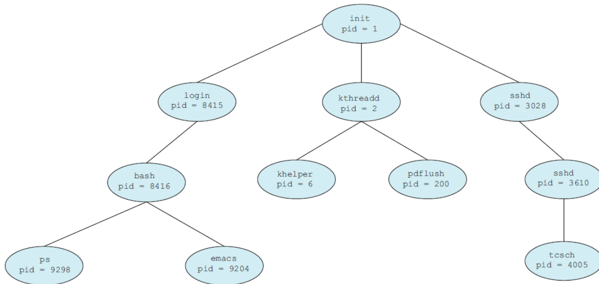
Process Creation

- A process may create several new processes.
  - The creating process: the parent process.
  - The new processes: the children processes.

Process Creation

- A process may create several new processes.
  - The creating process: the parent process.
  - The new processes: the children processes.

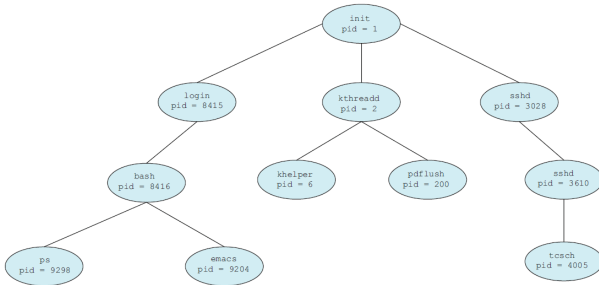
- These processes are forming a tree of processes.



Process Creation

- A process may create several new processes.
  - The creating process: the parent process.
  - The new processes: the children processes.

- These processes are forming a tree of processes.




```
# it lists complete information for all active processes in the system
ps -el
```

Parent-Child Resource Sharing Options

- Parent and children share all resources.

Parent-Child Resource Sharing Options

- Parent and children share all resources.
- Children share subset of parent's resources.

Parent-Child Resource Sharing Options

- Parent and children share all resources.
- Children share subset of parent's resources.
- Parent and child share no resources.
  - The child obtains the required resources directly from the OS.

Parent-Child Execution Options

- The parent continues to execute concurrently with its children.

Parent-Child Execution Options

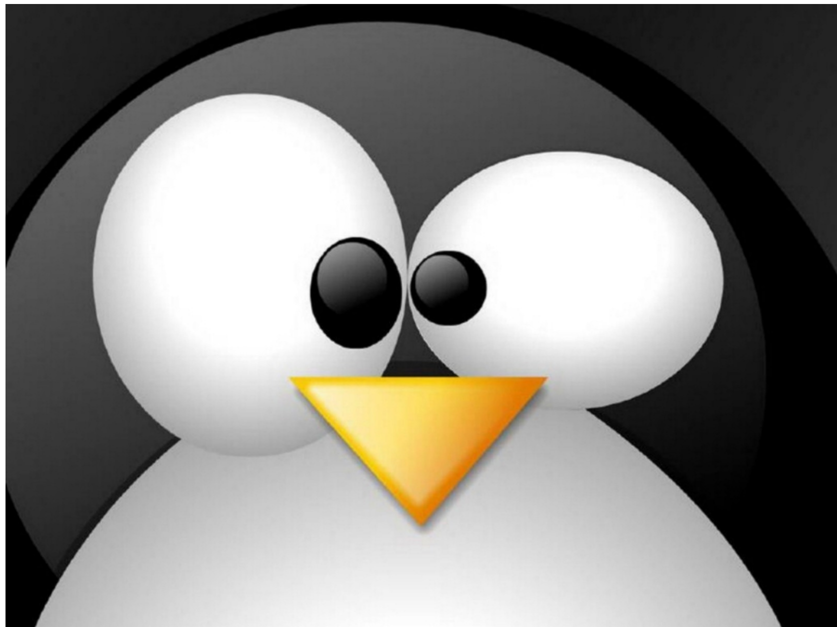
- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

Parent-Child Address Space Options

- The child process is a duplicate of the parent process (it has the same program and data as the parent).

Parent-Child Address Space Options

- The child process is a duplicate of the parent process (it has the same program and data as the parent).
- The child process has a new program loaded into it.



Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)

Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)
  - A system call loads a binary program into memory.

Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)
  - A system call loads a binary program into memory.
  - Replacing the previous contents of the address space.

Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)
  - A system call loads a binary program into memory.
  - Replacing the previous contents of the address space.
  - Begins execution of the new program.

Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)
  - A system call loads a binary program into memory.
  - Replacing the previous contents of the address space.
  - Begins execution of the new program.

- Creating a new process (`fork()`)

Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)
  - A system call loads a binary program into memory.
  - Replacing the previous contents of the address space.
  - Begins execution of the new program.

- Creating a new process (`fork()`)
  - The new process (child) initially is a near-duplicate of its parent process.

Running a New Process in Linux (1/2)

- Executing a new program (`exec()`)
  - A system call loads a binary program into memory.
  - Replacing the previous contents of the address space.
  - Begins execution of the new program.

- Creating a new process (`fork()`)
  - The new process (child) initially is a near-duplicate of its parent process.
  - Often, the new process immediately executes a new program.

Running a New Process in Linux (2/2)

- Executing a new program in a new process:
  1. First, a `fork` to create a new process,
  2. and then an `exec` to load a new binary into that process.

The Exec Family of Calls (1/3)

- There is no single exec function.
- The simplest of these calls is execl().
  - It replaces the current process image with a new one.

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
```

The Exec Family of Calls (2/3)

- Example

```
int ret;

ret = execl("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);

if (ret == -1)
  perror("execl");
```

The Exec Family of Calls (3/3)

- `l` and `v`: arguments are provided via a list or an array (vector).
- `p`: the user's full path is searched.
- `e`: a new environment is supplied for the new process.

```
#include <unistd.h>
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Creating a New Process (1/4)

- The only way to create a new process.
- The new process (child) running the same image as the current one (parent).
- Both processes continue to run, if nothing special had happened.

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

pid_t fork(void);
```

Creating a New Process (2/4)

- This `fork()` function is called once, but it returns twice.
  - The PID of the new child $\rightarrow$ to the parent.
  - 0 $\rightarrow$ to the child.

Creating a New Process (2/4)

- This `fork()` function is called once, but it returns twice.
  - The PID of the new child $\rightarrow$ to the parent.
  - 0 $\rightarrow$ to the child.
- The child and the parent process are identical, except for:

Creating a New Process (2/4)

- This `fork()` function is called once, but it returns twice.
  - The PID of the new child $\rightarrow$ to the parent.
  - 0 $\rightarrow$ to the child.
- The child and the parent process are identical, except for:
  - Their PIDs and their parents' PID.

Creating a New Process (2/4)

- This `fork()` function is called once, but it returns twice.
  - The PID of the new child $\rightarrow$ to the parent.
  - 0 $\rightarrow$ to the child.
- The child and the parent process are identical, except for:
  - Their PIDs and their parents' PID.
  - Reset resource statistics at the child.

Creating a New Process (2/4)

- This `fork()` function is called once, but it returns twice.
  - The PID of the new child $\rightarrow$ to the parent.
  - 0 $\rightarrow$ to the child.
- The child and the parent process are identical, except for:
  - Their PIDs and their parents' PID.
  - Reset resource statistics at the child.
  - Clear any pending signals at the child.

Creating a New Process (2/4)

- This `fork()` function is called once, but it returns twice.
  - The PID of the new child → to the parent.
  - 0 → to the child.
- The child and the parent process are identical, except for:
  - Their PIDs and their parents' PID.
  - Reset resource statistics at the child.
  - Clear any pending signals at the child.
  - The acquired file locks are not inherited by the child.

Creating a New Process (3/4)

- Example

```
pid_t pid = fork();

if (pid == -1) {
  perror("fork");
  exit(1);
}

if (pid > 0)
  printf("I am the parent of pid = %d!\n", pid);
else
  printf("I am the child!\n");
```

Creating a New Process (4/4)

- Example

```
pid_t pid = fork();

if (pid == -1) {
  perror("fork");
  exit(1);
}

if (pid == 0) { // the child
  const char *args[] = {"windlass", NULL};
  int ret;
  ret = execv("/bin/windlass", args);
  if (ret == -1) {
    perror("execv");
    exit(1);
  }
}
```

Process Termination

Process Termination (1/4)

- Process executes last statement and then asks the OS to delete it using the `exit()` system call.

Process Termination (1/4)

- Process executes last statement and then asks the OS to delete it using the `exit()` system call.
  - Returns status data from the child to the parent via `wait()`.

Process Termination (1/4)

- Process executes last statement and then asks the OS to delete it using the `exit()` system call.
  - Returns status data from the child to the parent via `wait()`.
  - Process resources are deallocated by the OS.

Process Termination (2/4)

- A parent may terminate the execution of its children via `abort()`.

Process Termination (2/4)

- A parent may terminate the execution of its children via `abort()`.
- Some reasons for doing so:
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - The parent is exiting and the OS does not allow a child to continue if its parent terminates.

Process Termination (3/4)

- Some OSs do not allow child to exists if its parent has terminated.
- If a process terminates, all its children must also be terminated.
  - Cascading termination: all children, grandchildren, etc. are terminated.
  - The termination is initiated by the OS.

Process Termination (4/4)

- The parent process may wait for termination of a child via `wait()`.

Process Termination (4/4)

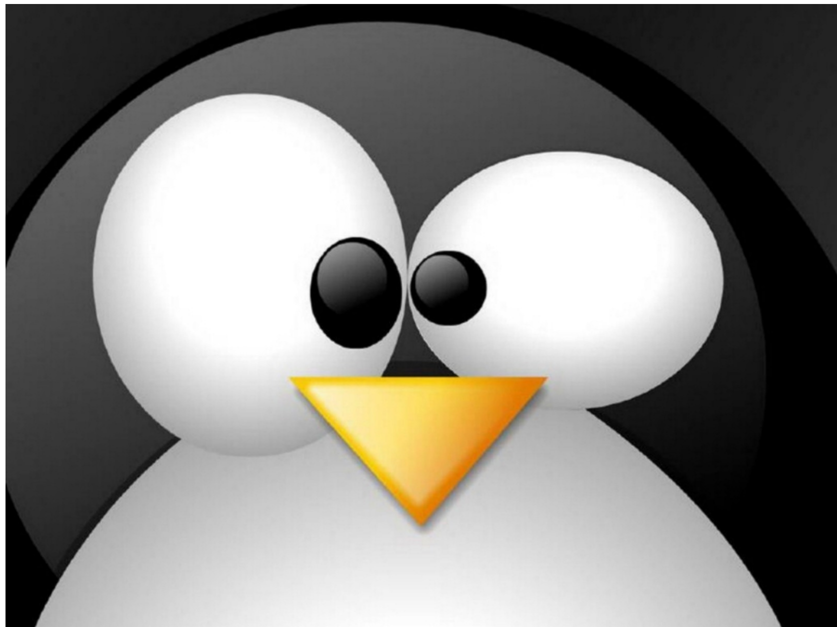
- The parent process may wait for termination of a child via `wait()`.
- The `wait()` returns the status information and the PID of the terminated process.

Process Termination (4/4)

- The parent process may wait for termination of a child via `wait()`.
- The `wait()` returns the status information and the PID of the terminated process.
- If a process has terminated, but whose parent has not yet called `wait()`, the process is a zombie.

Process Termination (4/4)

- The parent process may wait for termination of a child via `wait()`.
- The `wait()` returns the status information and the PID of the terminated process.
- If a process has terminated, but whose parent has not yet called `wait()`, the process is a zombie.
- If the parent terminated without invoking `wait()`, the process is an orphan.
  - In Linux, the `init` process becomes the parent of all orphans.



The `exit()` System Call

- The `exit()` performs some basic shutdown steps, then instructs the kernel to terminate the process.

- The `status` is used to denote the process's exit status.

```
#include <stdlib.h>

void exit(int status);
```

Waiting for Terminated Child Processes (1/3)

- When a child dies before its parent, the kernel puts the child into the zombie state.

Waiting for Terminated Child Processes (1/3)

- When a child dies before its parent, the kernel puts the child into the zombie state.
- Some basic kernel data structures containing potentially useful data is kept for the process.

Waiting for Terminated Child Processes (1/3)

- When a child dies before its parent, the kernel puts the child into the zombie state.
- Some basic kernel data structures containing potentially useful data is kept for the process.
- A process in this state waits for its parent to inquire about its status.

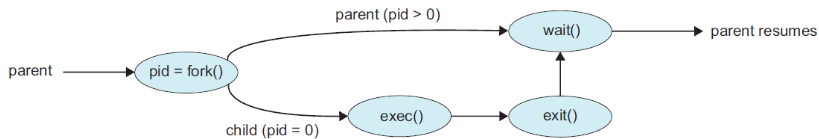
Waiting for Terminated Child Processes (1/3)

- When a child dies before its parent, the kernel puts the child into the zombie state.
- Some basic kernel data structures containing potentially useful data is kept for the process.
- A process in this state waits for its parent to inquire about its status.

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

pid_t wait(int *status);
```

Waiting for Terminated Child Processes (2/3)



Waiting for Terminated Child Processes (3/3)

```c
int main (void) {
  int status;
  pid_t pid;

  if (fork() == 0) return 1; // the child

  pid = wait(&status);

  if (pid == -1) perror("wait");

  printf("pid = %d\n", pid);

  return 0;
}
```

More on Launching and Waiting for Processes

- waitpid() to wait for a process with a known PID.

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

pid_t waitpid(pid_t pid, int *status, int options);
```

- Launching and waiting for a new process

```
#include <stdlib.h>

int system(const char *command);
```

Summary

Summary

- Process vs. Program

Summary

- Process vs. Program
- Process states: new, running, waiting, ready, terminated

Summary

- Process vs. Program
- Process states: new, running, waiting, ready, terminated
- Process Control Block (PCB)

Summary

- Process vs. Program
- Process states: new, running, waiting, ready, terminated
- Process Control Block (PCB)
- Process scheduling: scheduling queues, context switching

Summary

- Process vs. Program
- Process states: new, running, waiting, ready, terminated
- Process Control Block (PCB)
- Process scheduling: scheduling queues, context switching
- Process operations: creation (parent-child), termination

Questions?