3. Process Concept

[ECE321/ITP302] Operating Systems

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore inter-process communication using shared memory and message passing
- To describe communication in client-server systems

Agenda

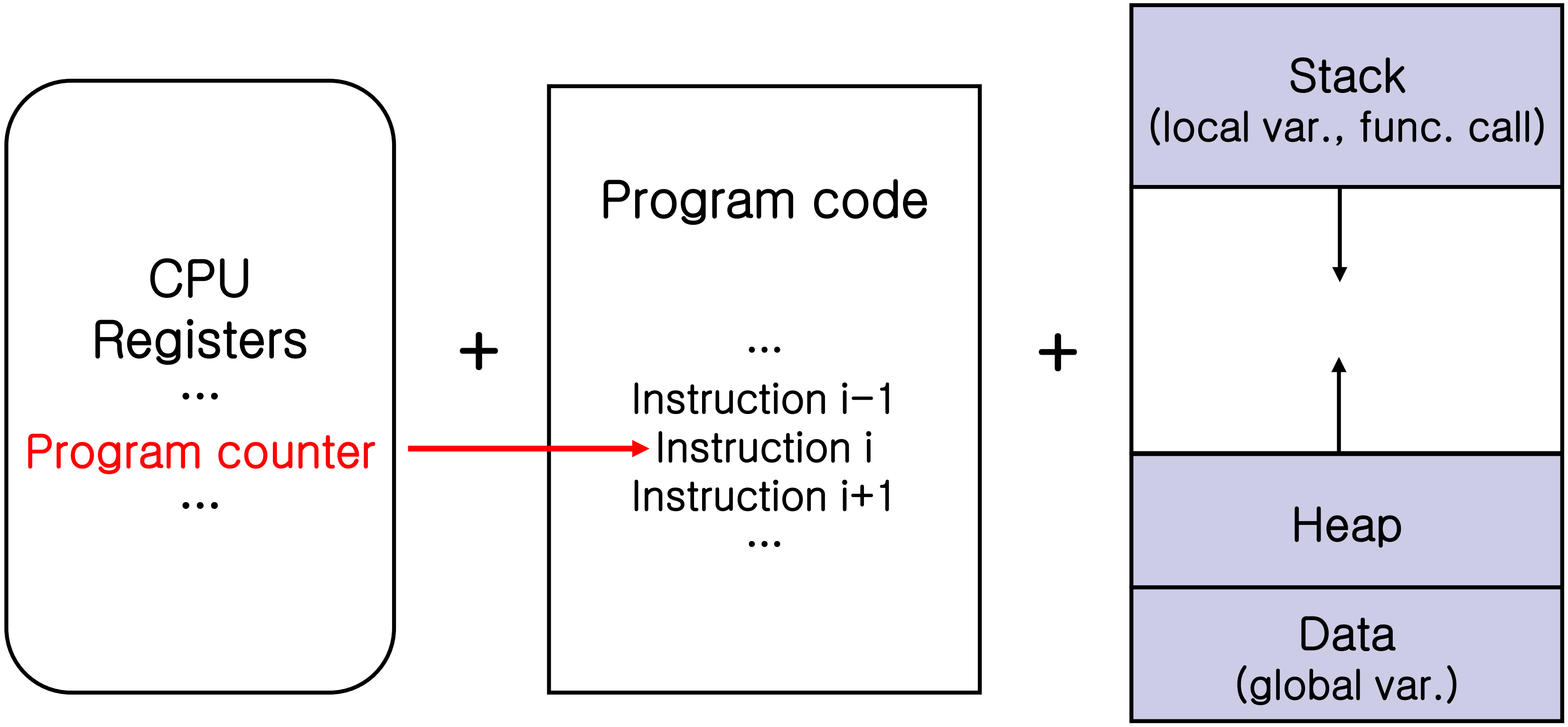
- Overview
- Process scheduling
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
- Program is *passive* entity stored on disk (executable file), process is *active*
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

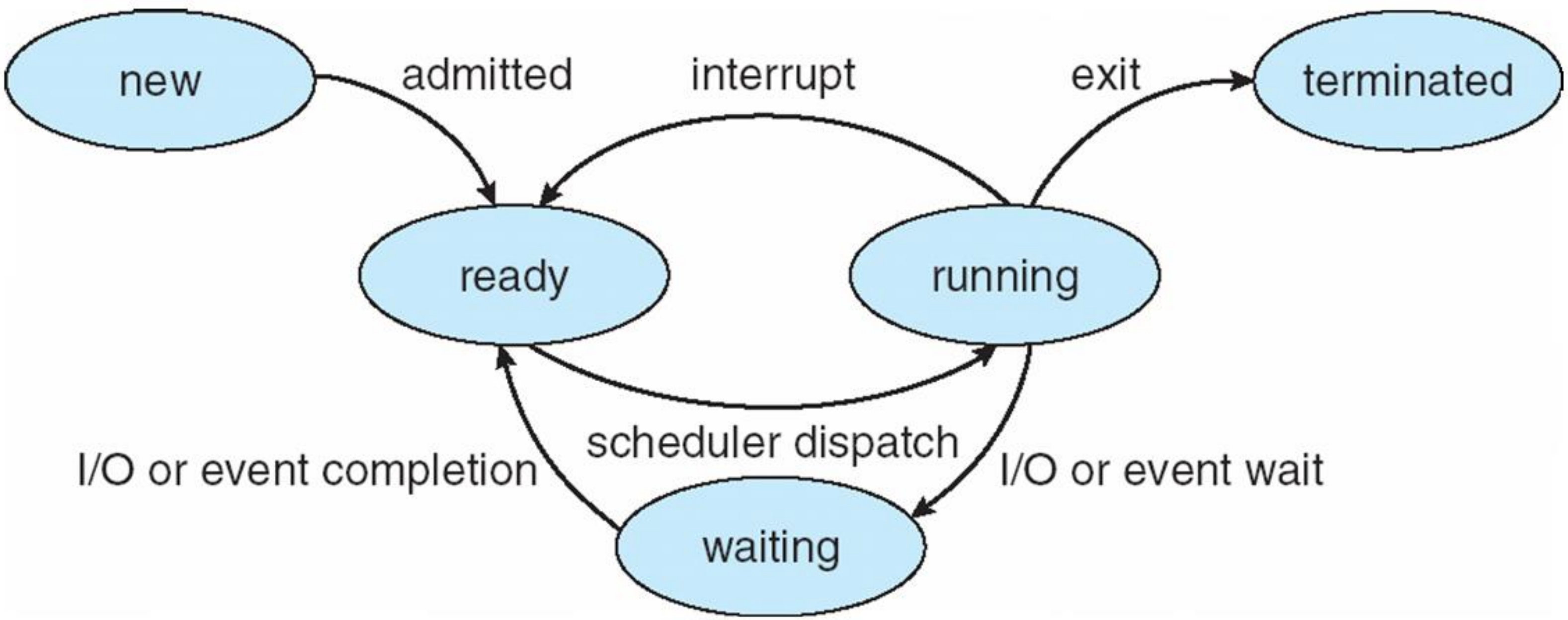
Process

- Process = program in execution + resource

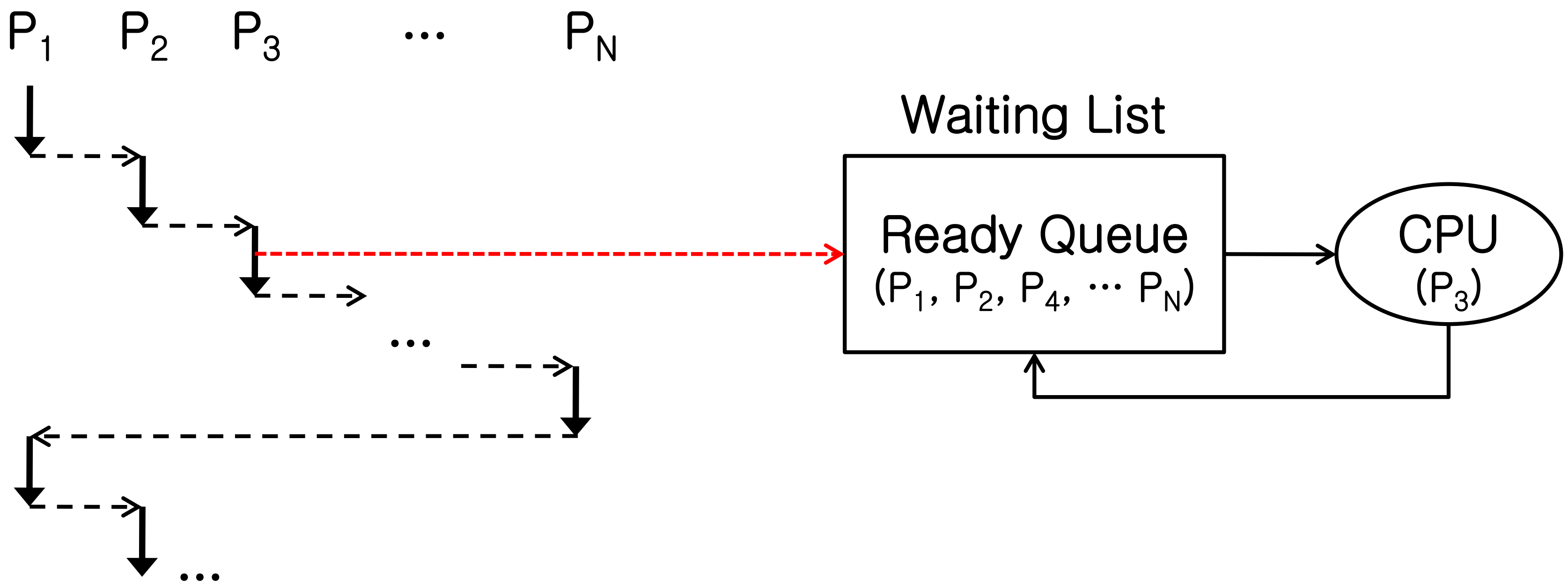


Process State

- **New**: being created
- **Running**: in execution
  - Only one process can be running on a processor at any time
- **Ready**: waiting to be assigned to a processor
- **Waiting**: waiting for some event to occur
- **Terminated**

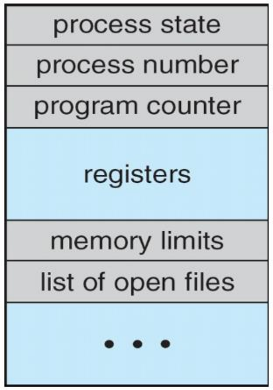


Ready/Running State



Process Control Block (PCB)

- OS manages processes using PCB
  - Process Control Block (PCB)
    repository for any information about process



Contents	Examples
Process state	new, ready, running, waiting, terminated, …
Process number	pid (Process ID)
CPU Registers	program counter (address of next instruction to execute) accumulator, general registers, stack pointer, …
CPU Scheduling info.	priority, pointer to queue, …
Memory-management info.	base and limit registers, page/segment table, …
Accounting info.	CPU-time used, time limits, account #, …
I/O status info.	List of open files, I/O devices allocated

Agenda

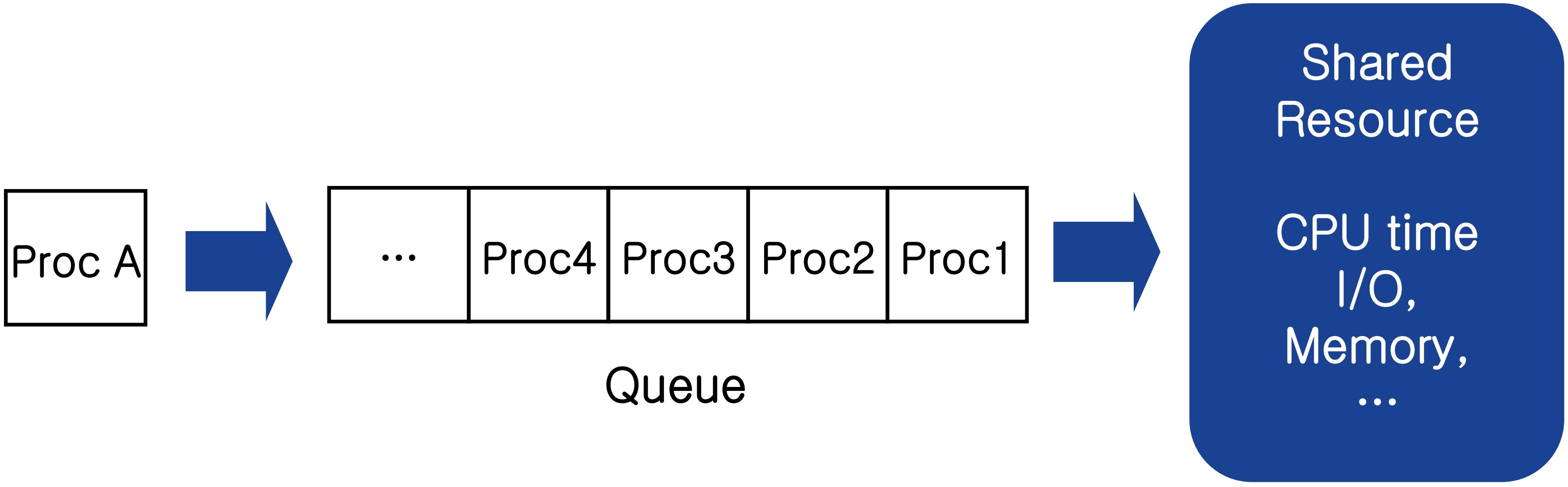
- Overview
- <u>Process scheduling</u>
- Operations on processes
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Process Scheduling

- Scheduling: assigning tasks to a set of resources
- Process scheduling: selecting a process to execute on CPU
  - Only one process can run on each processor at a time.
  - Other processes should wait
- Objectives of scheduling
  - Maximize CPU utilization (multiprogramming)
  - Users can interact with each program (time sharing)

Scheduling Queue

- Scheduling queue: waiting list of processes for CPU time or other resources

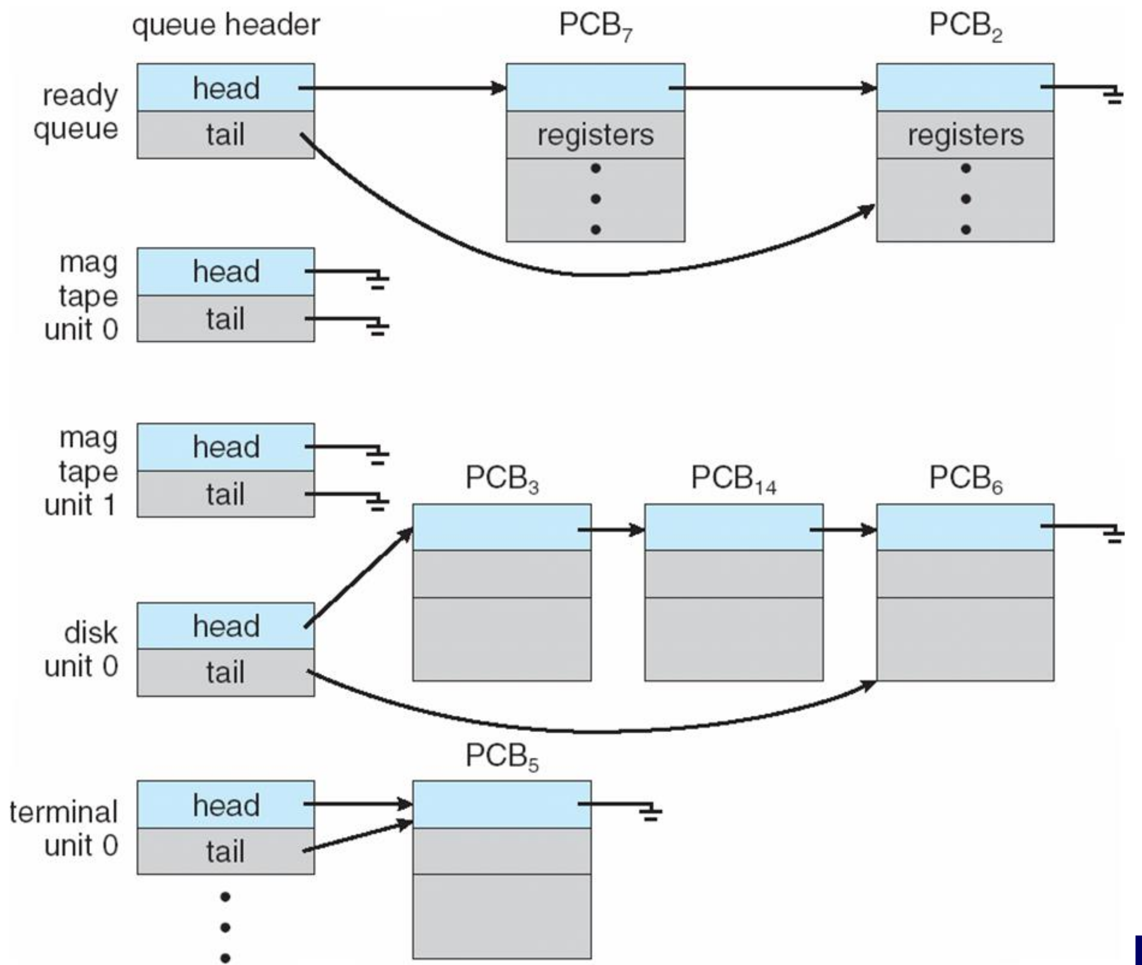


Types of Scheduling Queues

- Job queue
  - List of all processes in system
- Ready queue
  - List of processes, residing in main memory, ready to execute
- Device queue
  - List of processes waiting for a particular I/O device
  - Each device has its own device queue

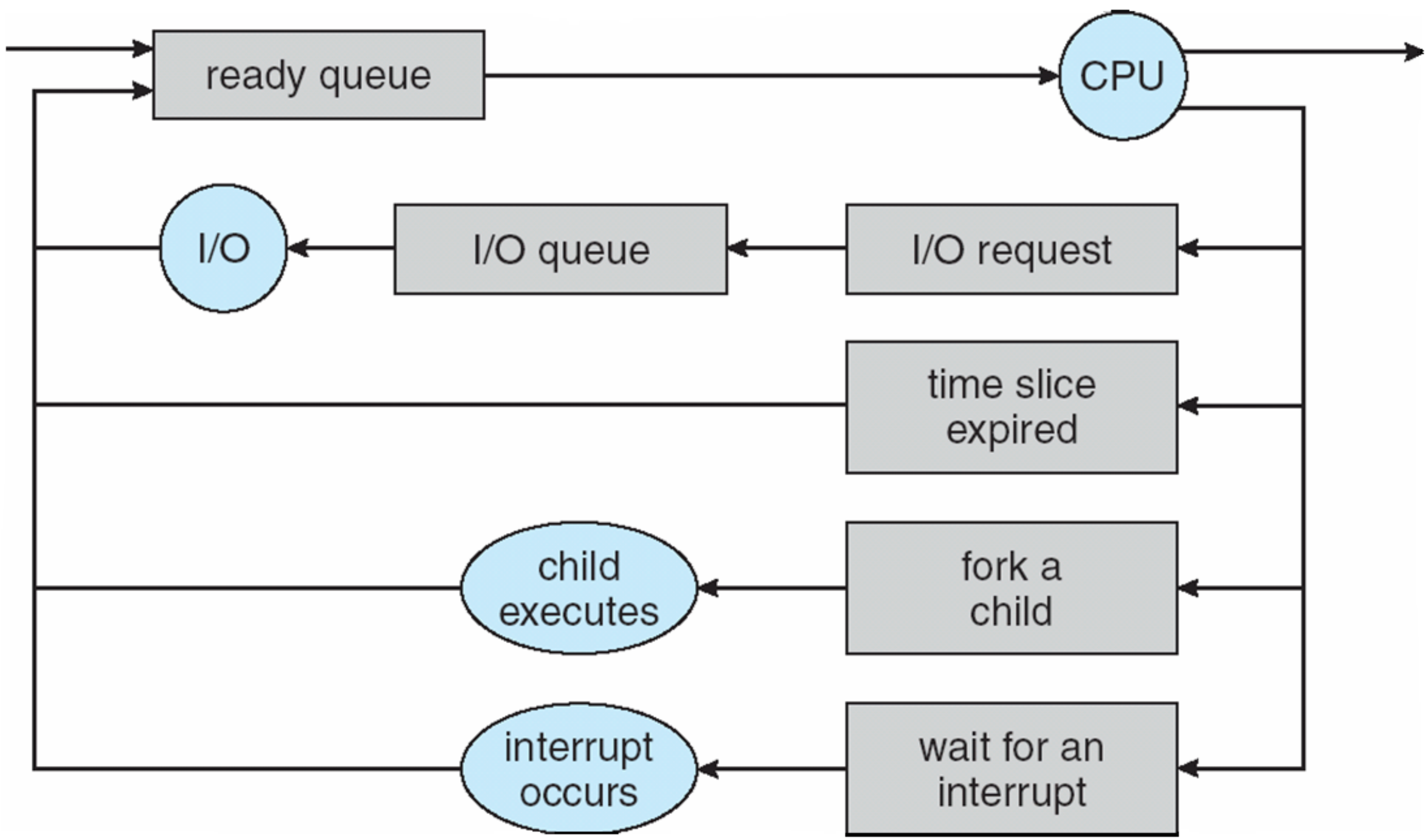
Scheduling Queue

- Each queue is usually represented by linked list



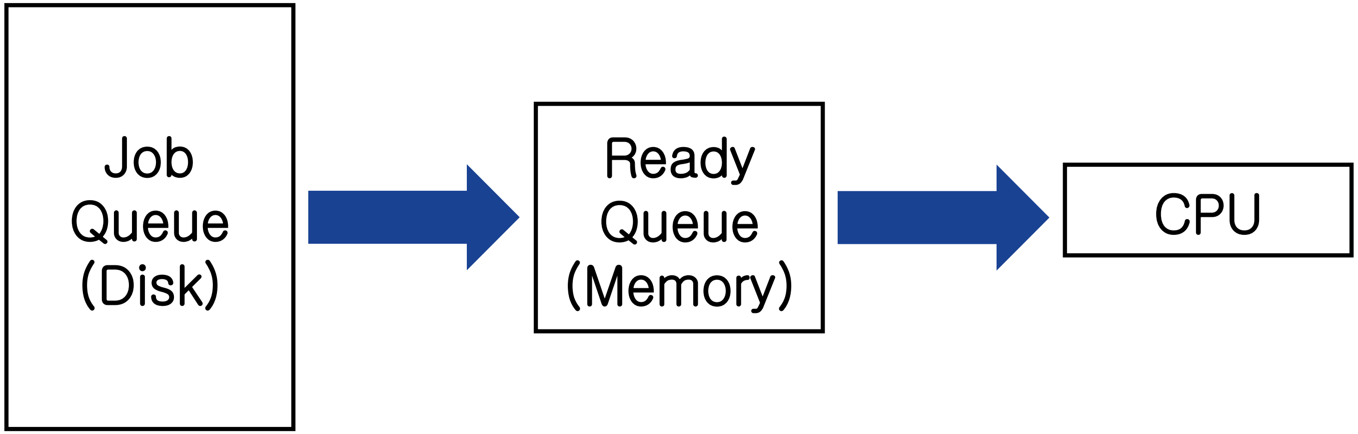
Queueing Diagram

- Representation of process scheduling
  - A process migrates among various scheduling queues throughout its lifetime



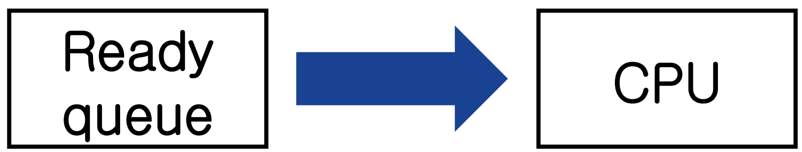
Schedulers

- Scheduler selects processes from queues in some fashion
  - Long-term scheduler (job scheduler)
  - Short-term scheduler (CPU scheduler)



Schedulers

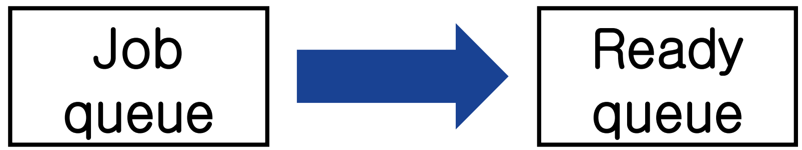
- Short-term scheduler (CPU scheduler)



  - Executed frequently (at least once every 100 msec.)
  - Scheduling time should be very short

Schedulers

- **Long-term scheduler (job scheduler)**

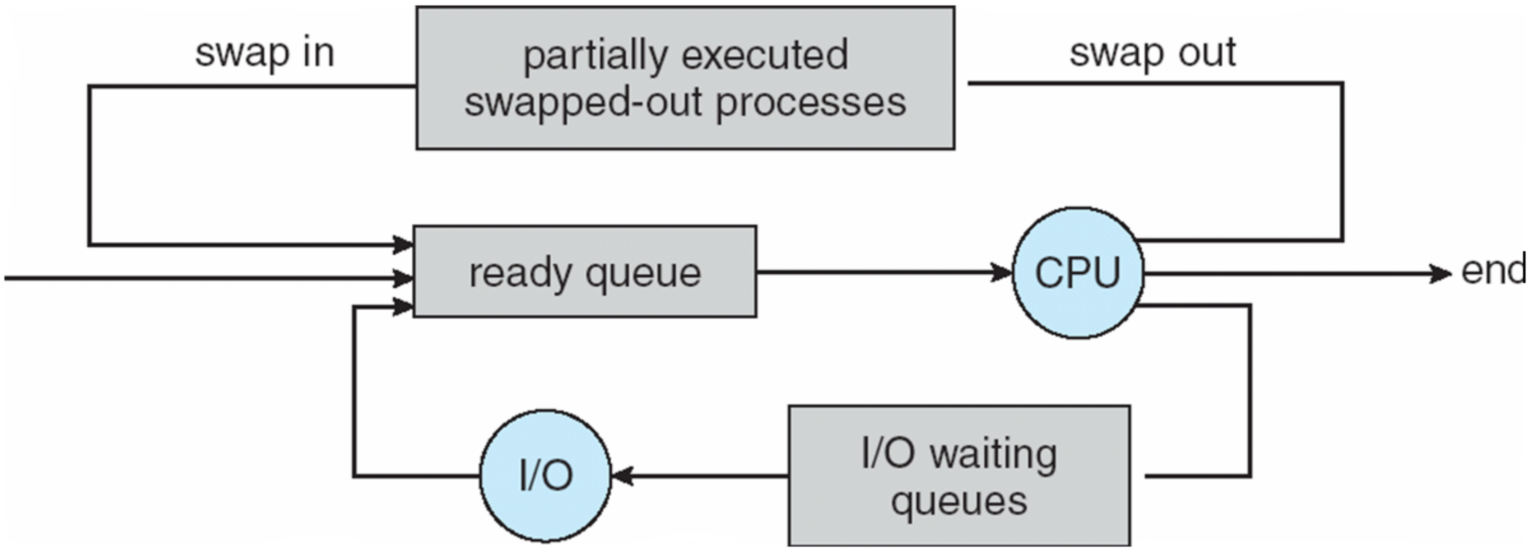


  - Controls **degree of multiprogramming**
    - In stable state, average process creation rate == average process departure rate
  - Executed less frequently
    - Executed only when a process leaves the system
  - Processes can be described as either:
    - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
    - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
  - Hopefully, long-term scheduler should select a good mix of **I/O-bound** and **CPU-bound** processes

Schedulers

- In some systems, long-term scheduler may be absent or minimal
  Ex) UNIX, Windows
  - System stability depends on physical limitation or self-adjusting nature of human

- Some time-sharing system has medium-term scheduler
  - Reduce degree of multiprogramming by removing processes from memory



Multitasking in Mobile Systems

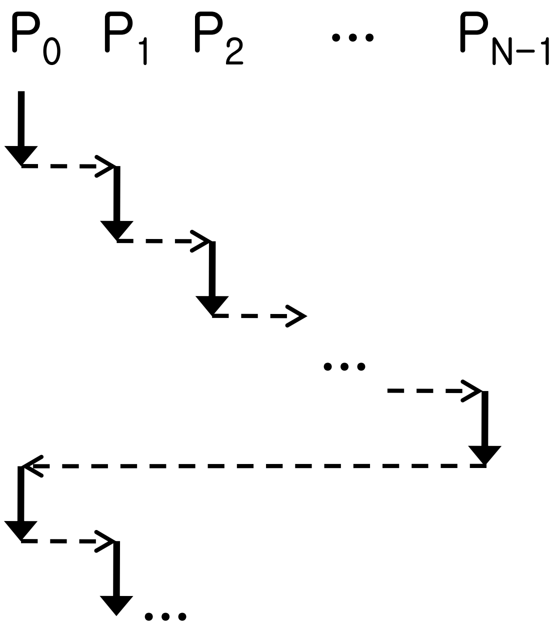
- **Some systems / early systems allow only one process to run, others suspended**

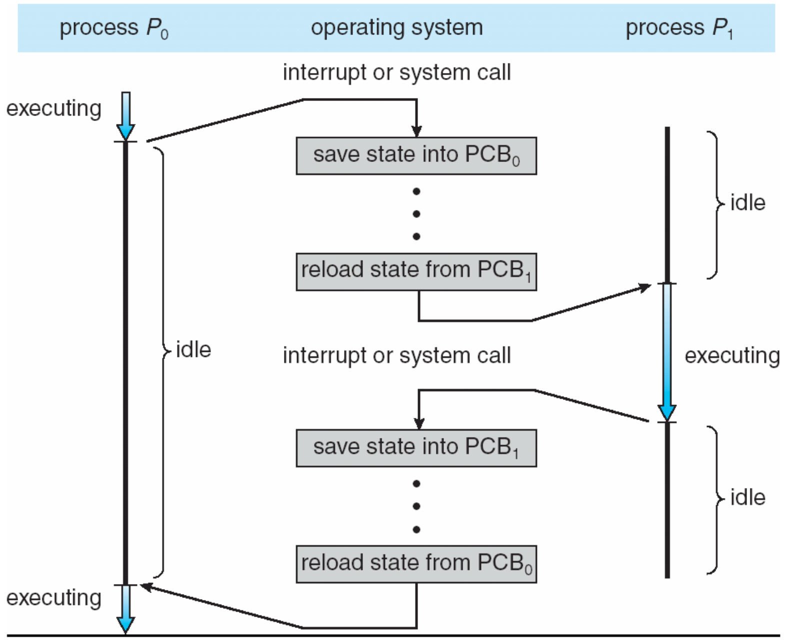
- **Due to screen real estate, user interface limits iOS provides for a**
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

- **Android runs foreground and background, with fewer limits**
  - Background process uses a **service** to perform tasks
  - **A service**: a separate application component that runs on behalf of the background process.
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Context Switch

- Switching running process requires context switch
  - State save of current process (PCB)
  - State restore of different process





Context Switch

- Context switch: the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource.

- "Context" includes
  - Register contents
  - OS specific data
    - Extra data required by advanced memory-management technique
    Ex) page table, segment table, …

- When to switch?
  - Multitasking
  - Interrupt handling

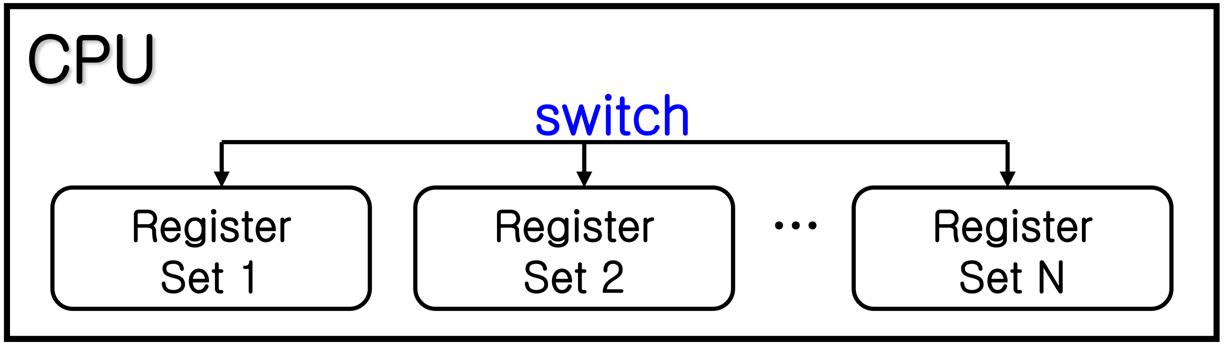
Context Switch

- Context switching requires considerable overhead.

- H/W supports for context-switching
  - H/W switching (single instruction to load/save all registers)

    cf. However, S/W switching can be more selective and save only that portion that actually needs to be saved and reloaded.
  - Multiple set of register for fast switching

    Ex) UltraSPARC



Agenda

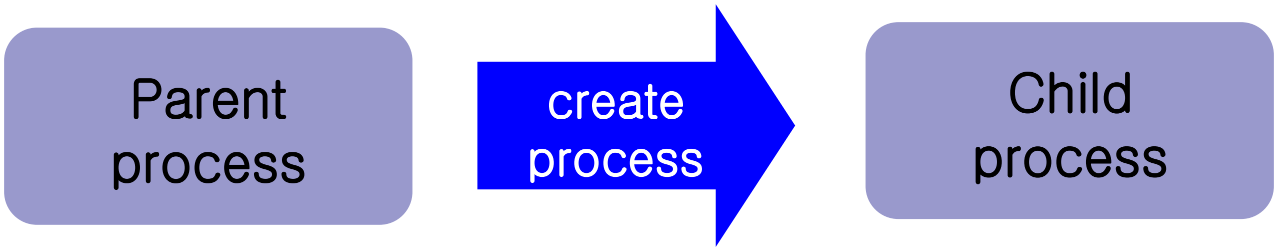
- Overview
- Process scheduling
- <u>**Operations on processes**</u>
- Inter-process communication
- Example of IPC system
- Communication in client-server systems

Operations on Processes

- Process create
- Process termination
- Process communication

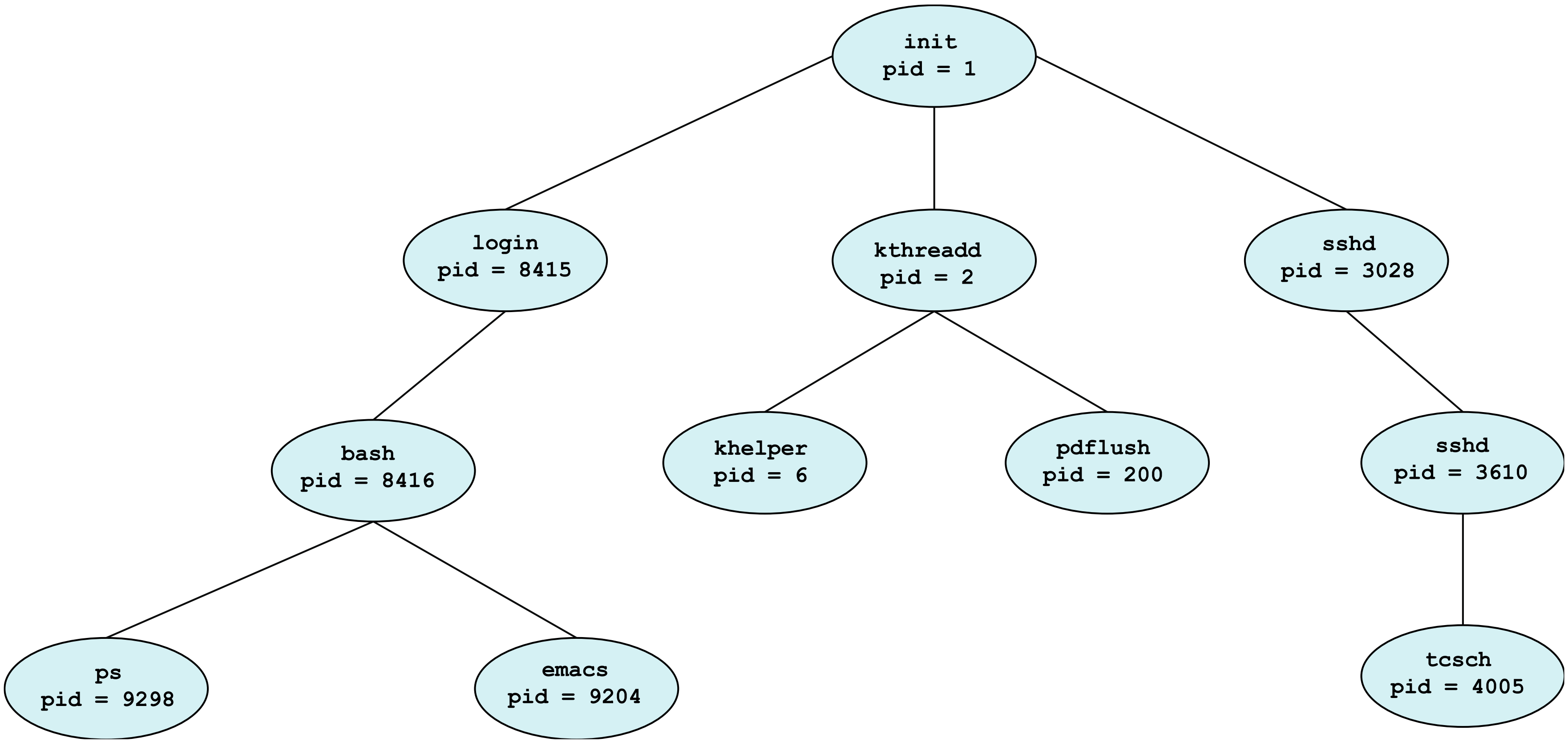
Process Creation

- Create-process system call
  - Create a process and assign a pid.



- Process tree
  - Parent-child relation between processes

A Tree of Processes in Linux



Displaying Process Information

- UNIX
  - ps [–el]
- Windows
  - Task manager (windows system program)
  - Process explorer (freeware)
    - Downloadable from FTP server

Process Creation

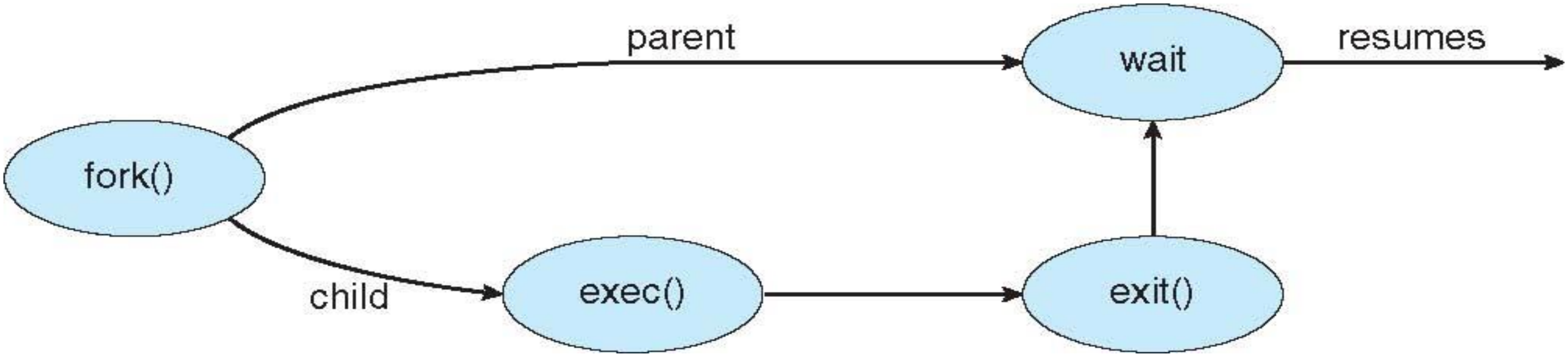
- Some options to create a process

	Options
Resource	1. Parent and children share all resources 2. Children share subset of parent's resources 3. Parent and child share no resources
Execution	1. Concurrent execution 2. Parent waits until child is terminated
Address space	1. Child duplicate of parent 2. Child has a program loaded into it

Process Creation in UNIX

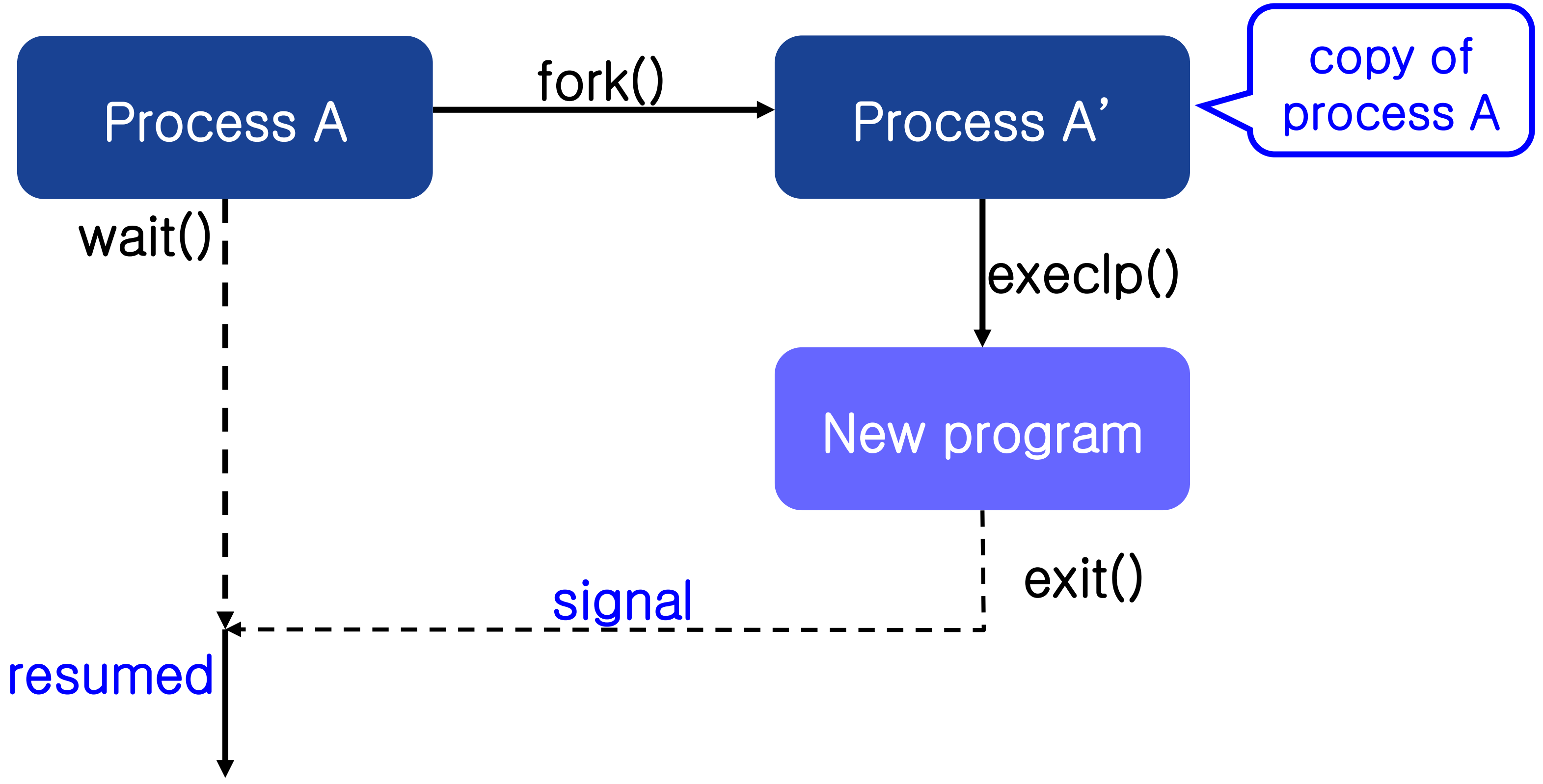
- UNIX system calls related to process creation
  - fork(): create process and returns its pid
    - In parent process, return value is pid of child
    - In child process, return value is zero
  - exec() family: execute a program. The new program substitutes the original one.
    - execl(), execv(), execlp(), execvp(), execle(), execve()
  - wait(): waits until child process is terminated

Process Creation



Example of Process Creation

- Executing other program



Example of Process Creation

```
int main()
{
   pid_t pid = fork();    // create a process
   if(pid < 0){           // error occurred
      fprintf(stderr, “fork failed\n”);
      exit(-1);
   } else if(pid == 0){   // child process
      execlp(“/bin/ls”, “ls”, NULL);
   } else {               // if pid != 0, parent process
      wait(NULL);         //  waits for child process to complete
      printf(“Child Completed\n”)
      exit(0);
   }
}
```

Example of Process Creation

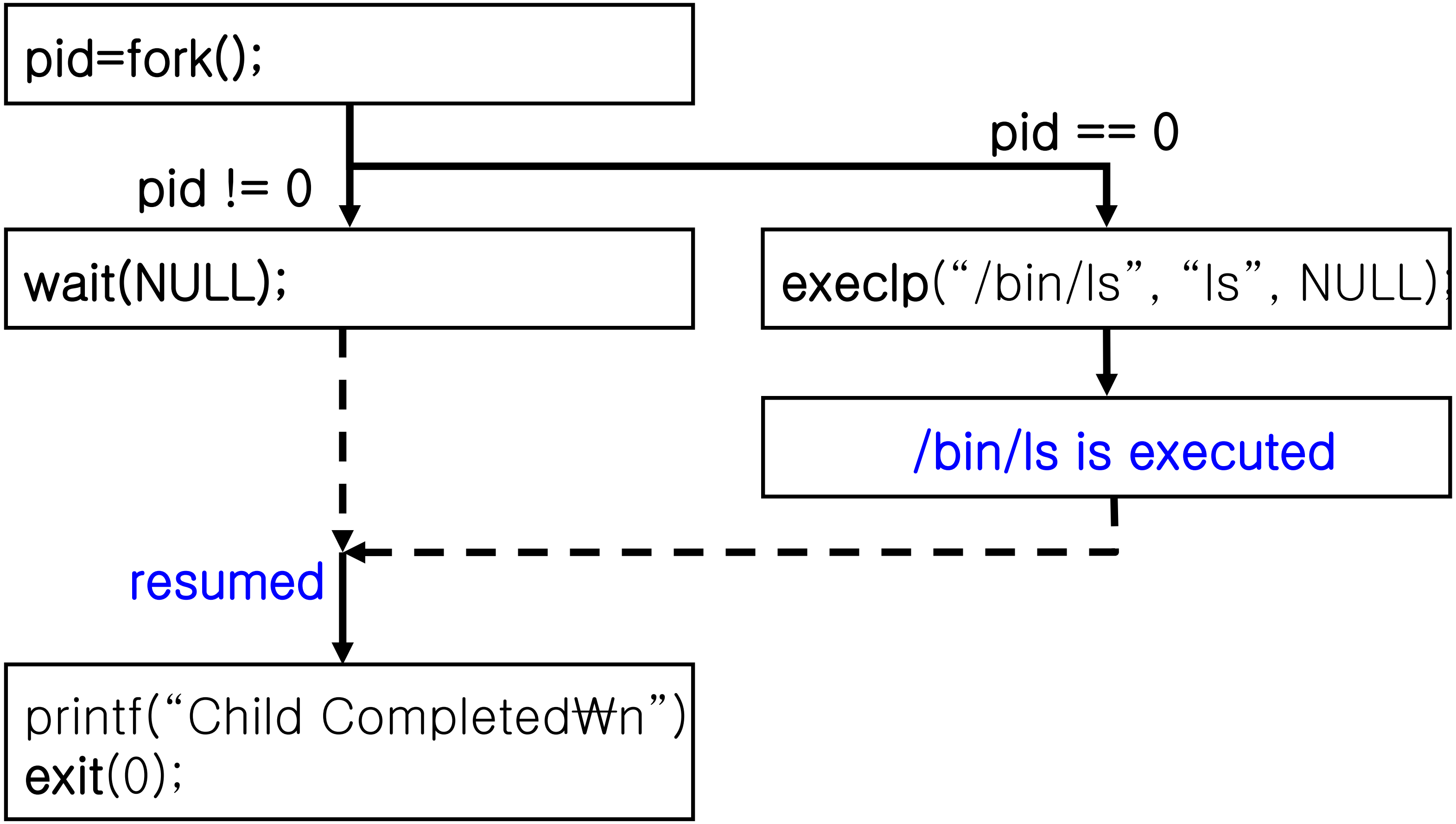
- **Parent process**

```
int main()
{
   pid_t pid = fork();
   if(pid < 0){
      fprintf(stderr, “fork failed\n”);
      exit(-1);
   } else if(pid == 0){
      execlp(“/bin/ls”, “ls”, NULL);
   } else {
      wait(NULL);
      printf(“Child Completed\n”)
      exit(0);
   }
}
```

- **Child process**

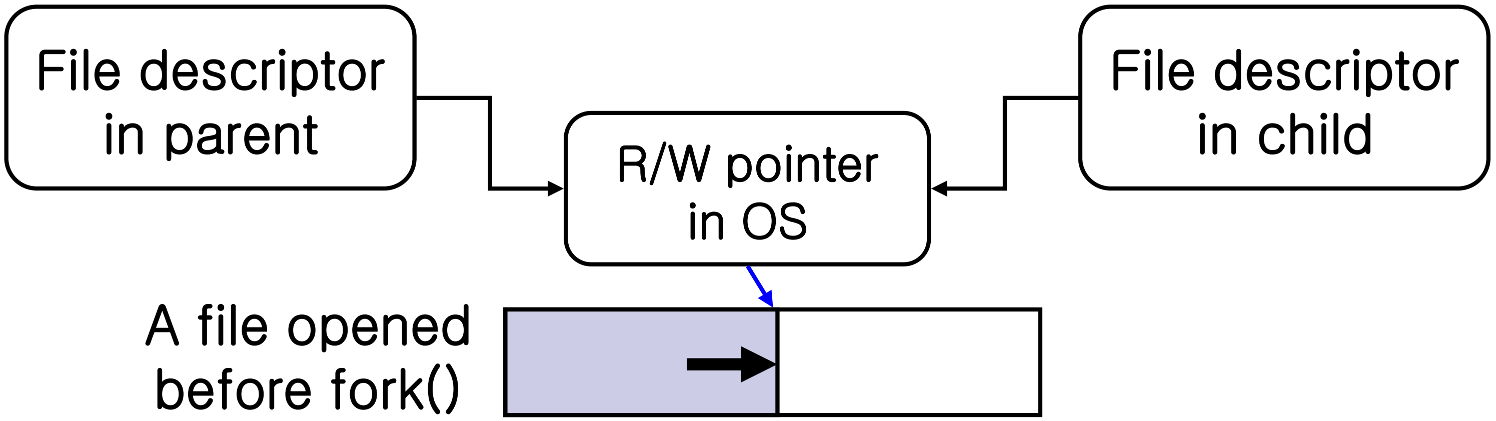
```
int main()
{
   pid_t pid = fork();
   if(pid < 0){
      fprintf(stderr, “fork failed\n”);
      exit(-1);
   } else if(pid == 0){
      execlp(“/bin/ls”, “ls”, NULL);
   } else {
      wait(NULL);
      printf(“Child Completed\n”)
      exit(0);
   }
}
```

Example of Process Creation



More About fork()

- Resource of child process
  - Data (variables): copies of variables of parent process
    - Child process has its own address space
    - The only difference is **pid** returned from **fork()**
  - Files
    - Opened before fork(): shared with parent
    - Opened after fork(): not shared



Process Creation in win32

- CreateProcess()
  - Similar to fork() of UNIX, but much more parameters to specify properties of child process
- WaitForSingleObject()
  - Similar to wait() of UNIX
- void ZeroMemory(PVOID *Destination*, SIZE_T *Length*);
  - Fills a block of memory with zeros.

For more detail, please refer MSDN homepage
(http://msdn.microsoft.com)

System Calls in win32

- Process creation

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

System Calls in win32

- Wait

```
DWORD WaitForSingleObject(
    HANDLE hHandle,        // pi.hProcess
    DWORD dwMilliseconds
);
```

Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Normal termination
  - exit(int return_code): invoked by child process
    - Clean-up actions
      - Deallocate memory
      - Close files
      - ETC.
    - return_code is passed to parent process
      - Usually, 0 means success
      - Parent can read the return code

```
int status = 0;
wait(&status);                  // wait until the child is terminated.
ret = WEXITSTATUS(status);      // return_code from the child
```

Abnormal Termination

- Possible reasons of abnormal termination
  - Child has exceeded allocated resource.
  - The task assigned child process is no longer required.
  - Parent is terminated, and OS doesn't allow child process to continue (ex: VMS).
    - Cascading termination

- System calls for process termination
  - Self-termination

    Ex) abort() of UNIX: Send SIGABRT signal to OS to make core dump.
  - Terminating other process

    Ex) TerminateProcess() of win32

Usually, such a system call can be invoked only by its parent process.