CS 332/532 Systems Programming

Lecture 18

-Process-

Professor : Mahmut Unan– UAB CS

Agenda

- Process management
- Unix Process
- Create a child process
- fork
- wait

Process Creation

Process spawning

- When the OS creates a process at the explicit request of another process

Parent process

- Is the original, creating, process

Child process

- Is the new process

Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Five-State Process Model

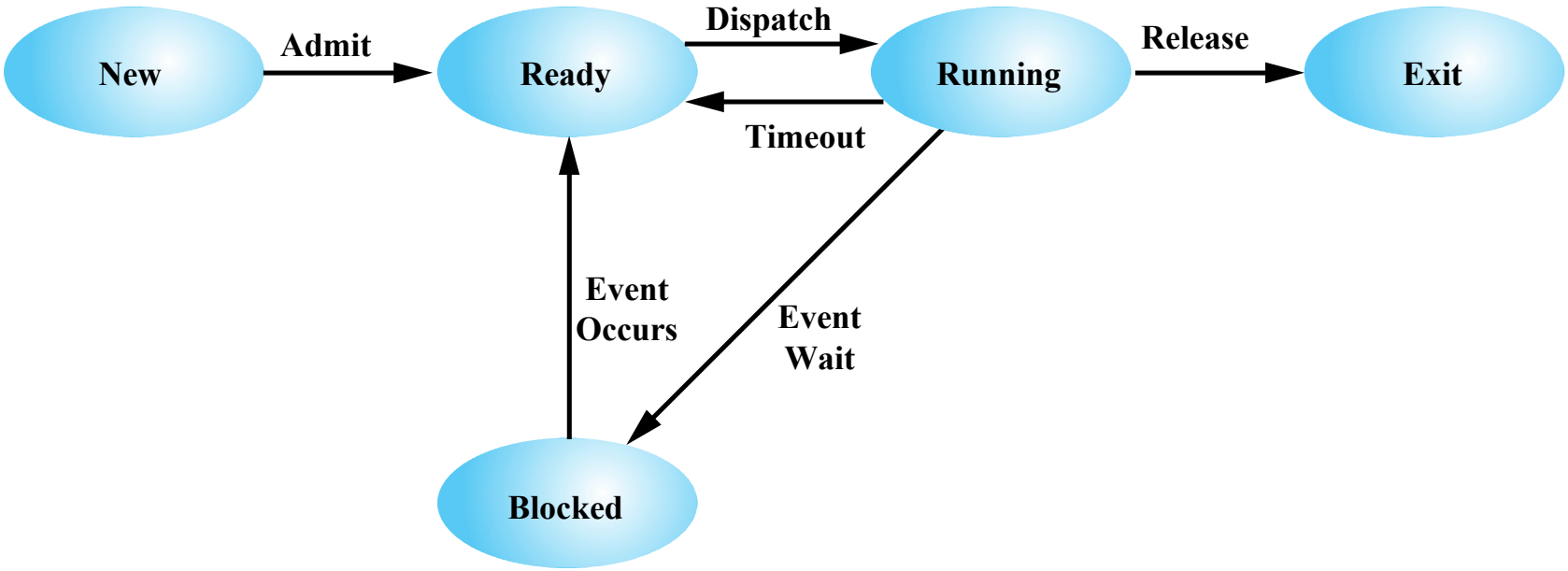
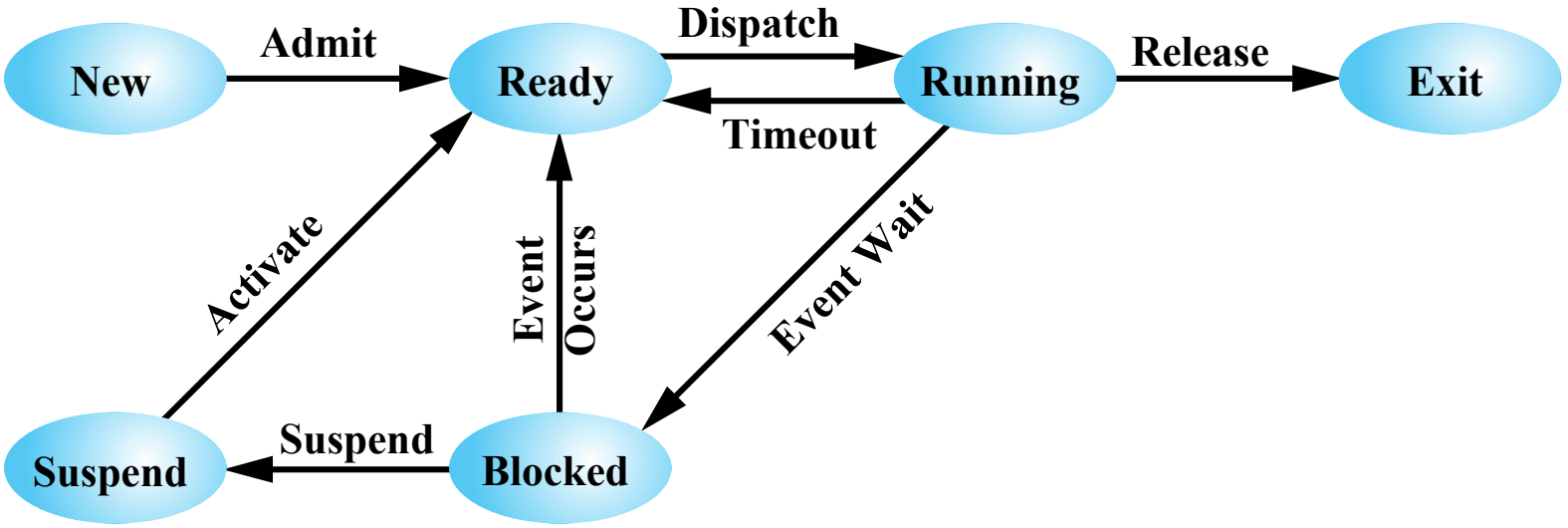
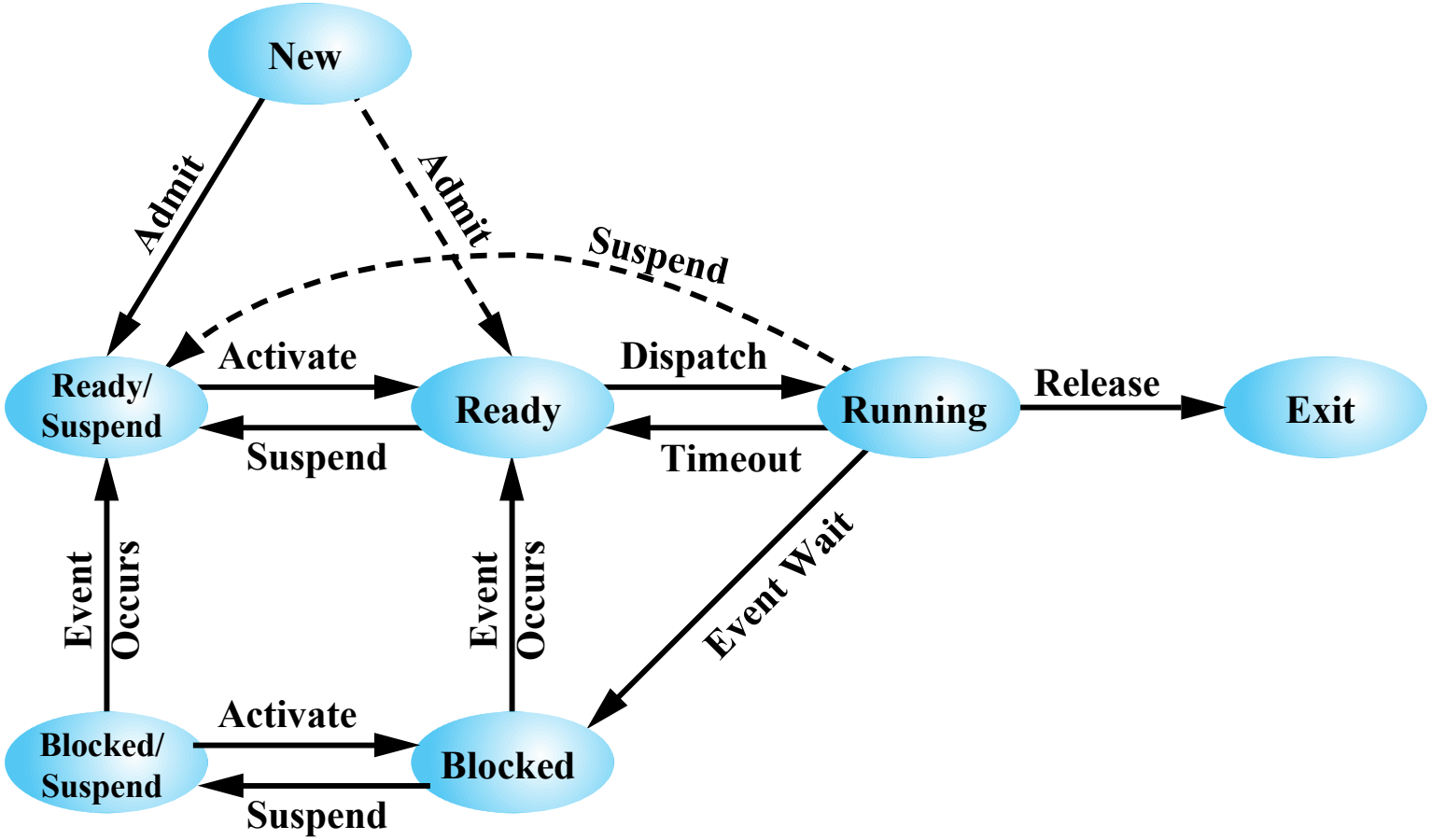


Figure 3.6 Five-State Process Model



(a) With One Suspend State

Figure 3.9 Process State Transition Diagram with Suspend States



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

Modes of Execution

User Mode

- Less-privileged mode
- User programs typically execute in this mode

System Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

Unix Processes

- The OS tracks processes through a five-digit ID number
  - **pid** or **process ID**.
- Each process in the system has a unique **pid**.
- If you want to list the running processes, use the **ps** (process status) command
  - to display the full option

    ```
    ps -f
    ```

    UID, PID, PPID, C, STIME, TTY, CMD, TIME…..

Unix Processes

- to stop a process
  - `kill`
  - `kill PID`
  - `kill -9 PID`
- Init Process
  - PID 1 and PPID 0
- Foreground Process
- Background Process

fork()

- fork - create a child process

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

https://man7.org/linux/man-pages/man2/fork.2.html

fork()

- We can use the fork() system call to create a new process (referred to as the child process) which is an identical image of the calling process (referred to as the parent process).
- Here is the C interface for the fork() system call:

```
#include <unistd.h>

pid_t fork(void);
```

fork()

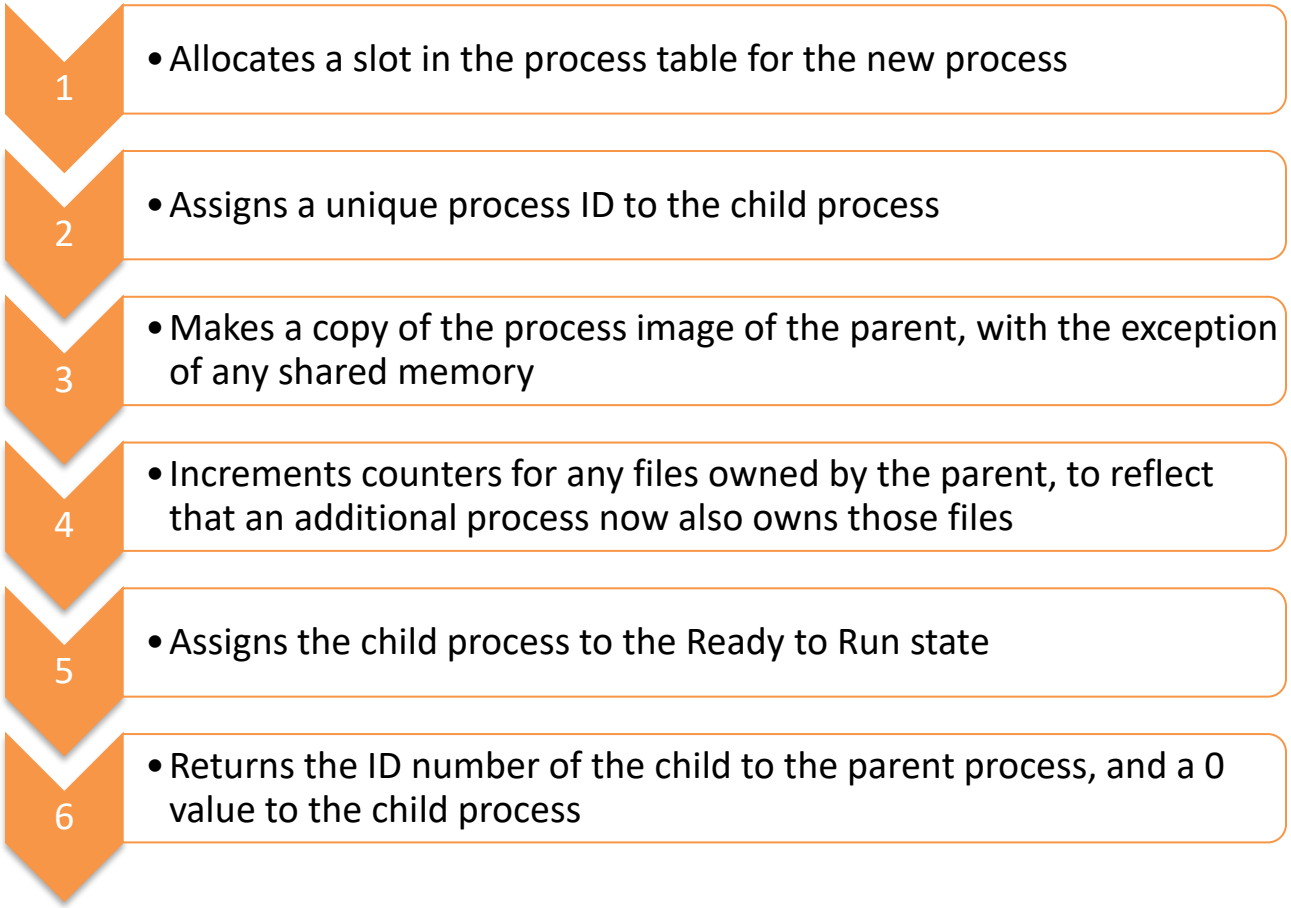
- If the fork() call is successful, then it returns the process ID of the child process to the parent process and returns 0 in the child process.
- fork() returns a negative value in the parent process and sets the corresponding errno variable (external variable defined in *errno.h*) if there is any error in process creation and the child process is not created.
- We can use perror() function (defined in *stdio.h*) to print the corresponding system error message. Look at the man page for perror to find out more about the perror() function.

fork()

- Once the parent process creates the child process, the parent process continues with its normal execution.
- If the parent process exits before the child process completes its execution and terminates, the child process will become a zombie process (*i.e.*, a process without a parent process).
- Alternatively, the parent process could wait for the child process to terminate using the wait() function.
- The wait() system call will suspend the execution of the calling process until one of the child process terminates and if there are no child processes available the wait() function returns immediately.

Process Control

- Process creation is by means of the kernel system call, *fork()*
- When a process issues a fork request, the OS performs the following functions:



Hello World!

```c
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

Hello World!

```c
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

```c
int main() {
    fork();
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
Hello, World!
```

Hello World!

```c
int main() {
    int processId = fork();
    printf("Hello, World! from= %d\n",processId);
    return 0;
}
```

```
Hello, World! from= 96037
Hello, World! from= 0
```

```c
int main() {
    printf("This is before the fork statement\n");
    fork();
    printf("After the FIRST fork\n");
    fork();
    printf("After the SECOND fork \n");
    fork();
    printf("After the THIRD fork \n");
    return 0;
}
```

```
After the FIRST fork
After the SECOND fork
After the FIRST fork
After the THIRD fork
After the SECOND fork
After the SECOND fork
After the SECOND fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
```

```c
int main() {

    fork();
    fork();
    fork();
    fork();
    printf("4 forks will work 16 times\n");
    return 0;
}
```

```
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
```

getpid()

```c
int main() {

    printf("before calling the fork %d\n",getpid());
    fork();
    printf("after calling the  FIRST fork %d\n",getpid());
    fork();
    printf("after calling the  SECOND fork %d\n",getpid());
}
```

```
before calling the fork 96717
after calling the  FIRST fork 96717
after calling the  SECOND fork 96717
after calling the  FIRST fork 96718
after calling the  SECOND fork 96719
after calling the  SECOND fork 96718
after calling the  SECOND fork 96720
```

wait()

wait, waitpid, waitid - wait for process to change state

```
 pid_t wait(int *wstatus);
 pid_t waitpid(pid_t pid, int *wstatus, int
options);
 int waitid(idtype_t idtype, id_t id, siginfo_t
*infop, int options);
```

https://www.man7.org/linux/man-pages/man2/waitid.2.html

wait()

- The wait() call returns the PID of the child process that terminated when successful, otherwise, it returns -1.
- The wait() call also sets an integer value that is passed as an argument to the function which can be inspected with various macros provided in <sys/wait.h> to determine how the child process completed (e.g., terminated normally, terminated by a signal).

wait() waitpid()

- If the calling process created more than one child process, we can use the waitpid() system call to wait on a specific child process to change state.
- A state change could be any one of the following events: the child was terminated; the child was stopped by a signal; or the child was resumed by a signal. Similar to wait(), waitpid() returns the PID of the child process that changed state when successful, otherwise, it returns -1.

wait() waitpid()

- Here are the C APIs for
  the wait() and waitpid() system calls:

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

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

```c
int main() {
    int processId = fork();
    int count;
    fflush(stdout);

    if (processId==0){
        count=1;
    }else{
        count=6;
    }
    if (processId!=0){
        wait();
    }
   int i;
    for (i=count;i<count+5;i++){
        printf("%d",i);
        fflush( stdout );
    }

}
```

```
12345678910
Process finished with exit code 0
```

https://www.youtube.com/watch?v=tcYo6hipaSA&ab_channel=CodeVault

Example 1

- We will create a sample program to illustrate how to use fork() to create a child process, wait for the child process to terminate, and display the parent and child process ID in both processes.

fork.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) { /* this is child process */
        printf("This is the child process, my PID is %ld and my parent PID is %ld\n",
            (long)getpid(), (long)getppid());
    } else if (pid > 0) { /* this is the parent process */
        printf("This is the parent process, my PID is %ld and the child PID is %ld\n",
            (long)getpid(), (long)pid);

        printf("Wait for the child process to terminate\n");
```

fork.c

```
18
19          printf("Wait for the child process to terminate\n");
20          wait(&status); /* wait for the child process to terminate */
21          if (WIFEXITED(status)) { /* child process terminated normally */
22              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23          } else { /* child process did not terminate normally */
24              printf("ERROR: Child process did not terminate normally!\n");
25              /* look at the man page for wait (man 2 wait) to
26                              determine how the child process was terminated */
27          }
28      } else { /* we have an error in process creation */
29          perror("fork");
30          exit(EXIT_FAILURE);
31      }
32
33      printf("[%ld]: Exiting program .....\n", (long)getpid());
34
35      return 0;
36  }
37
```

fork.c

```
(base) mahmutunan@MacBook-Pro lecture17 % ./exercise1
This is the parent process, my PID is 90695 and the child PID is 90696
Wait for the child process to terminate
This is the child process, my PID is 90696 and my parent PID is 90695
[90696]: Exiting program .....
Child process exited with status = 0
[90695]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture17 %
```