Processes

Jo, Heeseung

Today's Topics

What is the process?

How to implement processes?

What Is The Process?

Program?

vs.

Process?

vs.

Processor?

vs.

Task? Job?

Process Concept (1)

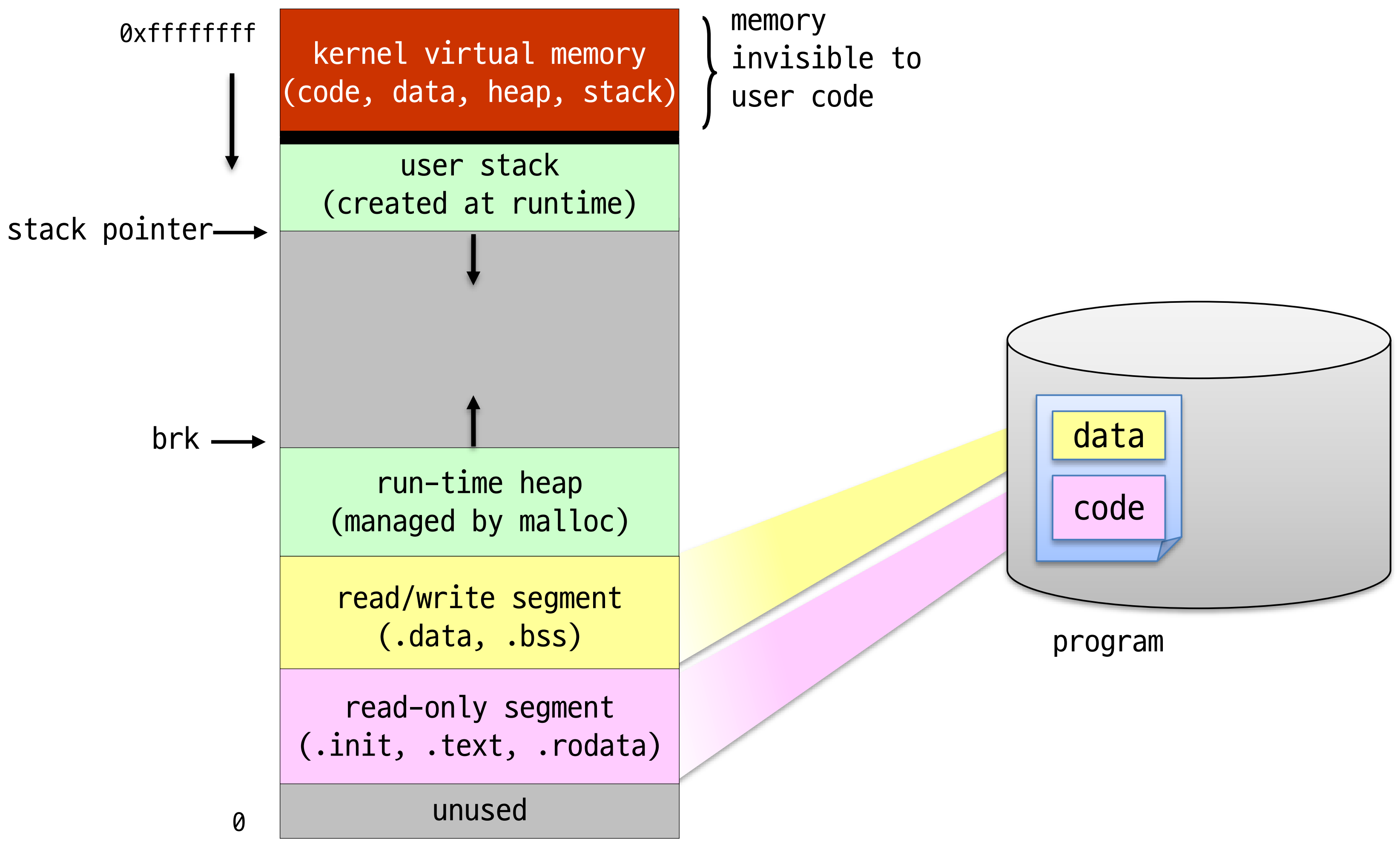
What is the process?

- An instance of a program in execution
- An encapsulation of the flow of control in a program
- A dynamic and active entity
- The basic unit of execution and scheduling
- A process is named using its process ID (PID)

- A process includes:
  - CPU contexts (registers)
  - OS resources (memory, open files, etc.)
  - Other information (PID, state, owner, etc.)

Process Concept (2)

Process in memory

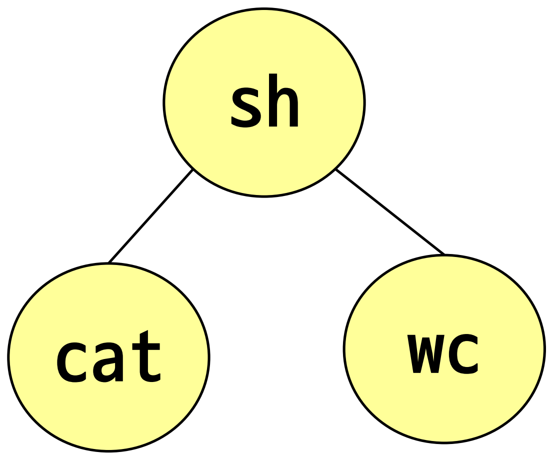


Process Creation (1)

Process hierarchy

- One process can create another process: parent-child relationship
- UNIX calls the hierarchy a "process group"
- Windows has no concept of process hierarchy
- Browsing a list of processes:
  - ps in UNIX
  - taskmgr (Task Manager) in Windows

```
$ cat file1 ¦ wc
```



Process Creation (2)

Process creation events

- Calling a system call
  - fork() in POSIX, CreateProcess() in Win32
  - Shells or GUIs use this system call internally
- System initialization
  - *init* process
  - PID 1 process

Process Creation (3)

Resource sharing

- Parent may inherit all or a part of resources and privileges for its children
  - UNIX: User ID, open files, etc.

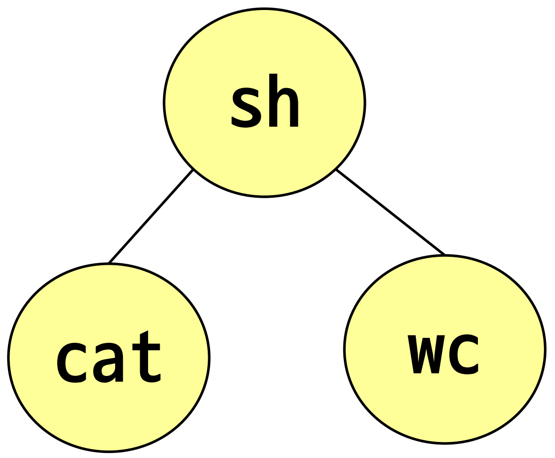
Execution

- Parent may either wait for it to finish, or it may continue in parallel

Address space

- Child duplicates the parent's address space or has a program loaded into it

```
$ cat file1 ¦ wc
```



Process Termination

Process termination events

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
  - Exceed allocated resources
  - Segmentation fault
  - Protection fault, etc.
- Killed by another process (involuntary)
  - By receiving a signal

```
#include <stdio.h>

int main()
{
    int i, fd;
    char buf[100];

    fd=open("a.txt", "r");
    if (fd==NULL)
        return -1;
    read(fd, buf, 1000);

    return 0;
}
```

fork()

fork() system call

- Creating a child process
- Copy the whole virtual address space of parent to create a child process
- Copy internal data structures to manage a child process

- Parent get the pid of a child
- Child get 0 value

fork()

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

int main()
{
	int pid;

	pid = fork();
	if (pid == 0)
	   /* child */
	   printf ("Child of %d is %d\n",
	                getppid(), getpid());
	else
	   /* parent */
	   printf ("I am %d. My child is %d\n",
	                getpid(), pid);
}
```

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

int main()
{
	int pid;

	pid = fork();
	if (pid == 0)
	   /* child */
	   printf ("Child of %d is %d\n",
	                getppid(), getpid());
	else
	   /* parent */
	   printf ("I am %d. My child is %d\n",
	                getpid(), pid);
}
```

fork(): Example Output

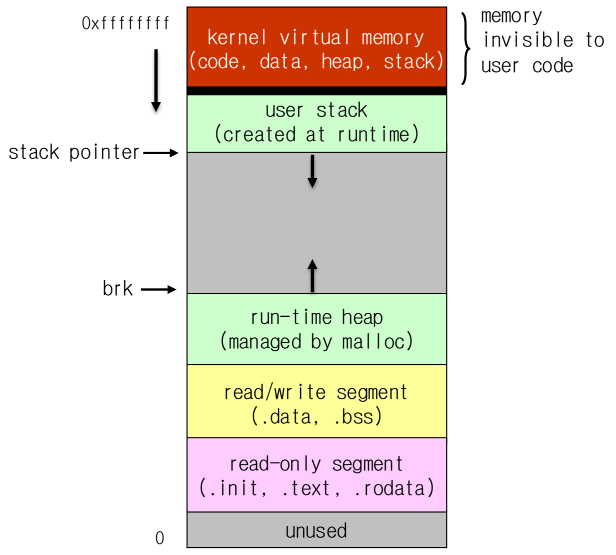
```
% ./a.out
I am 30000. My child is 30001.
Child of 30000 is 30001.

% ./a.out
Child of 30002 is 30003.
I am 30002. My child is 30003.
```

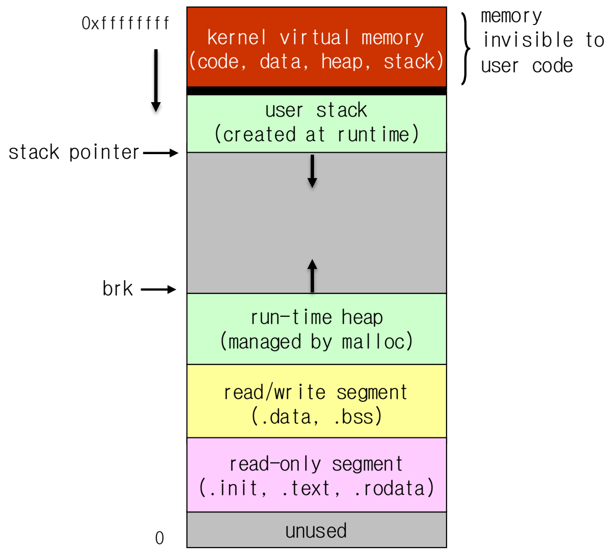
```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
     int pid;

     pid = fork();
     if (pid == 0)
        /* child */
        printf ("Child of %d is %d\n",
                 getppid(), getpid());
     else
        /* parent */
        printf ("I am %d. My child is %d\n",
                 getpid(), pid);
}
```

fork() and Virtual Address Space

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

int main()
{
     int pid;

     pid = fork();
     if (pid == 0)
        /* child */
        printf ("Child of %d is %d\n",
                 getppid(), getpid());
     else
        /* parent */
        printf ("I am %d. My child is %d\n",
                 getpid(), pid);
}
```

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

int main()
{
     int pid;

     pid = fork();
     if (pid == 0)
        /* child */
        printf ("Child of %d is %d\n",
                 getppid(), getpid());
     else
        /* parent */
        printf ("I am %d. My child is %d\n",
                 getpid(), pid);
}
```


0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
user stack (created at runtime)
stack pointer
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
read-only segment (.init, .text, .rodata)
0
unused

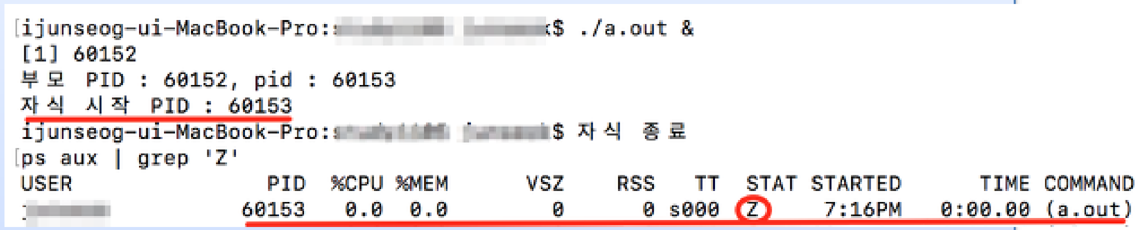

0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
user stack (created at runtime)
stack pointer
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
read-only segment (.init, .text, .rodata)
0
unused

Zombie vs. orphan process

Zombie process (defunct process)

- A process that completed execution (via the exit system call) but still has an entry in the process table
- This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status

```
int main() {

    pid_t childPid;

    childPid = fork();

    if (childPid > 0) {  // parent process
        printf("parent PID : %ld, pid : %d\n",(long)getpid(), childPid);
        sleep(30);
        printf("parent exit\n");
        exit(0);
    }
    else if (childPid == 0){  // 자식 코드
        printf("child PID : %ld\n", (long)getpid());
        sleep(1);
        printf("child exit\n");
        exit(0);
    }
    return 0;
}
```

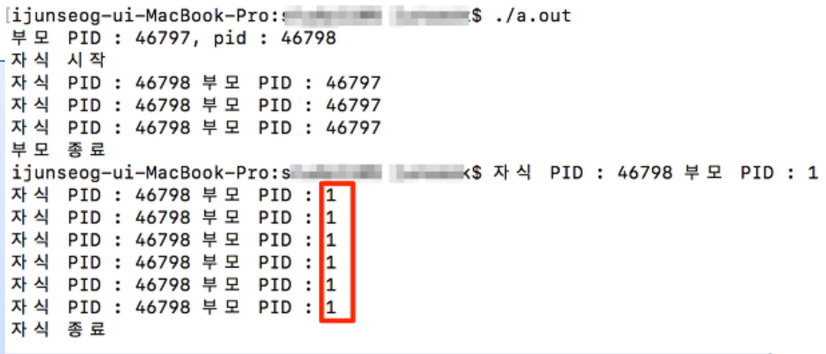


```
ijunseog-ui-MacBook-Pro:[redacted] [redacted]$ ./a.out &
[1] 60152
부모 PID : 60152, pid : 60153
자식 시작 PID : 60153
ijunseog-ui-MacBook-Pro:[redacted]$ 자식 종료
ps aux | grep 'Z'
USER                PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
[redacted]        60153   0.0  0.0        0      0 s000  Z     7:16PM   0:00.00 (a.out)
```

Zombie vs. orphan process

Orphan process

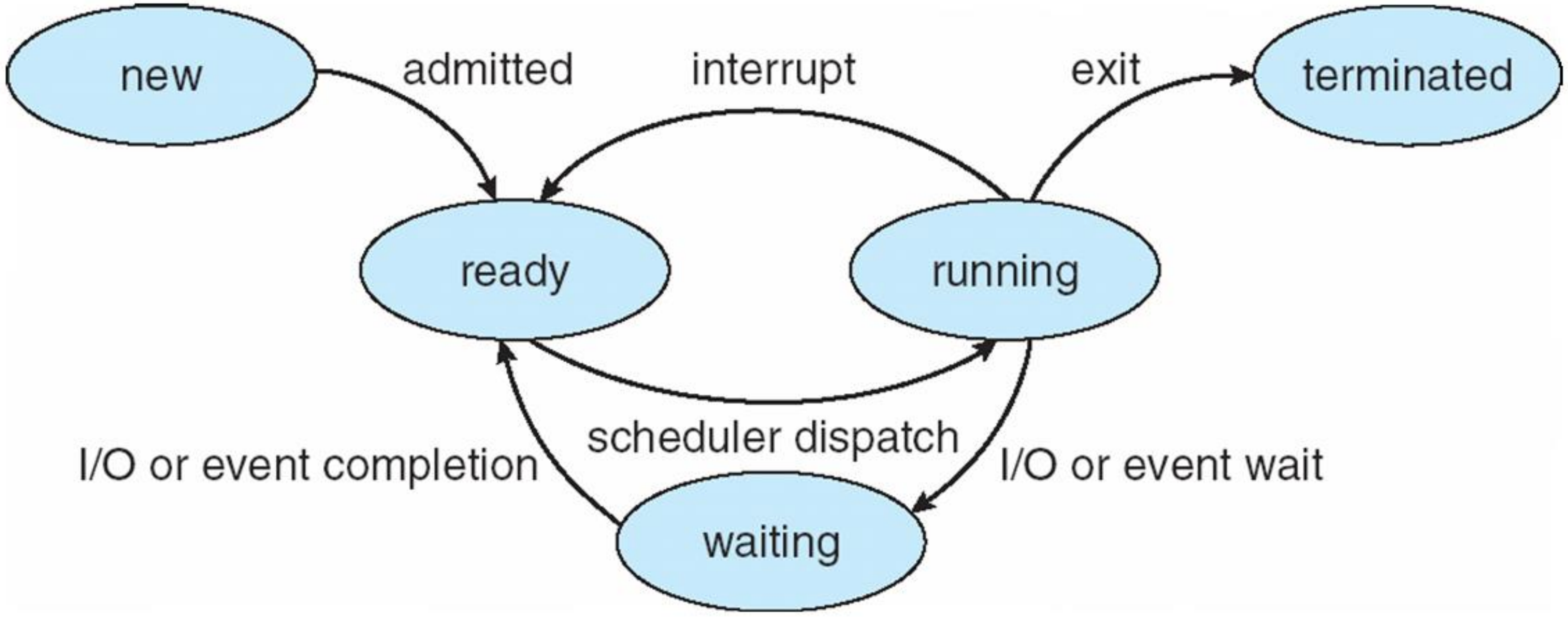
- A process whose parent process has finished or terminated, though it remains running itself
- Any orphaned process will be immediately adopted by the special init system process



```
ijunseog-ui-MacBook-Pro:[redacted]$ ./a.out
부모 PID : 46797, pid : 46798
자식 시작
자식 PID : 46798 부모 PID : 46797
자식 PID : 46798 부모 PID : 46797
자식 PID : 46798 부모 PID : 46797
부모 종료
ijunseog-ui-MacBook-Pro:[redacted]$ 자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 종료
```

```c
int main() {

    pid_t childPid;
    int i;

    childPid = fork();

    if (childPid > 0) {  // parent process
        printf("parent PID : %ld, pid : %d\n",(long)getpid(), childPid);
        sleep(2);
        printf("parent exit\n");
        exit(0);
    }
    else if (childPid == 0){  // child process
        for(i=0;i<10;i++) {
            printf("child PID : %ld parent PID : %ld\n",(long)getpid(), (long)getppid());
            sleep(1);
        }
        printf("child exit\n");
        exit(0);
    }
```

Process State Transition (1)



Process State Transition (2)

Linux example

```
root      991     1  0 Mar19 ?        Ss     0:03 nmbd -D
root      1019   934  0 Mar19 ?        S      0:00 smbd -F
root      1038     2  0 Mar19 ?        S      0:00 [hd-audio1]
root      1039     1  0 Mar19 tty4     Ss+    0:00 /sbin/getty -8 38400 tty4
root      1043     1  0 Mar19 tty5     Ss+    0:00 /sbin/getty -8 38400 tty5
root      1051     1  0 Mar19 tty2     Ss+    0:00 /sbin/getty -8 38400 tty2
root      1052     1  0 Mar19 tty3     Ss+    0:00 /sbin/getty -8 38400 tty3
root      1057     1  0 Mar19 tty6     Ss+    0:00 /sbin/getty -8 38400 tty6
root      1063     1  0 Mar19 ?        Ss     0:00 /usr/sbin/irqbalance
root      1130     1  0 Mar19 ?        Ss     0:00 cron
daemon    1131     1  0 Mar19 ?        Ss     0:00 atd
mysql     1186     1  0 Mar19 ?        Ssl    0:13 /usr/sbin/mysqld
root      1232     1  0 Mar19 ?        Sl     0:00 /usr/sbin/console-kit-daemon --no-daemon
root      1296     1  0 Mar19 ?        Ss     0:01 /usr/sbin/apache2 -k start
root      1380     1  0 Mar19 tty1     Ss+    0:00 /sbin/getty -8 38400 tty1
root      2840   401  0 Mar19 ?        S<     0:00 udevd --daemon
root      2930   401  0 Mar19 ?        S<     0:00 udevd --daemon
root      3011     2  0 Mar19 ?        S      0:00 [kdmflush]
www-data  3133  1296  0 Mar19 ?        S      0:00 /usr/sbin/apache2 -k start
root      5450     2  0 10:09 ?        S      0:00 [flush-8:0]
www-data  6470  1296  0 12:03 ?        S      0:00 /usr/sbin/apache2 -k start
www-data  6471  1296  0 12:03 ?        S      0:00 /usr/sbin/apache2 -k start
www-data  6580  1296  0 13:20 ?        S      0:00 /usr/sbin/apache2 -k start
www-data  6581  1296  0 13:20 ?        S      0:00 /usr/sbin/apache2 -k start
www-data  6584  1296  0 13:22 ?        S      0:00 /usr/sbin/apache2 -k start
www-data  7133  1296  0 14:16 ?        S      0:00 /usr/sbin/apache2 -k start
root      7142   953  0 14:24 ?        Ss     0:00 sshd: root@pts/0
root      7205  7142  0 14:24 pts/0    Ss     0:00 -bash
root      7261  7205  0 14:27 pts/0    R+     0:00 ps -ef ax
```

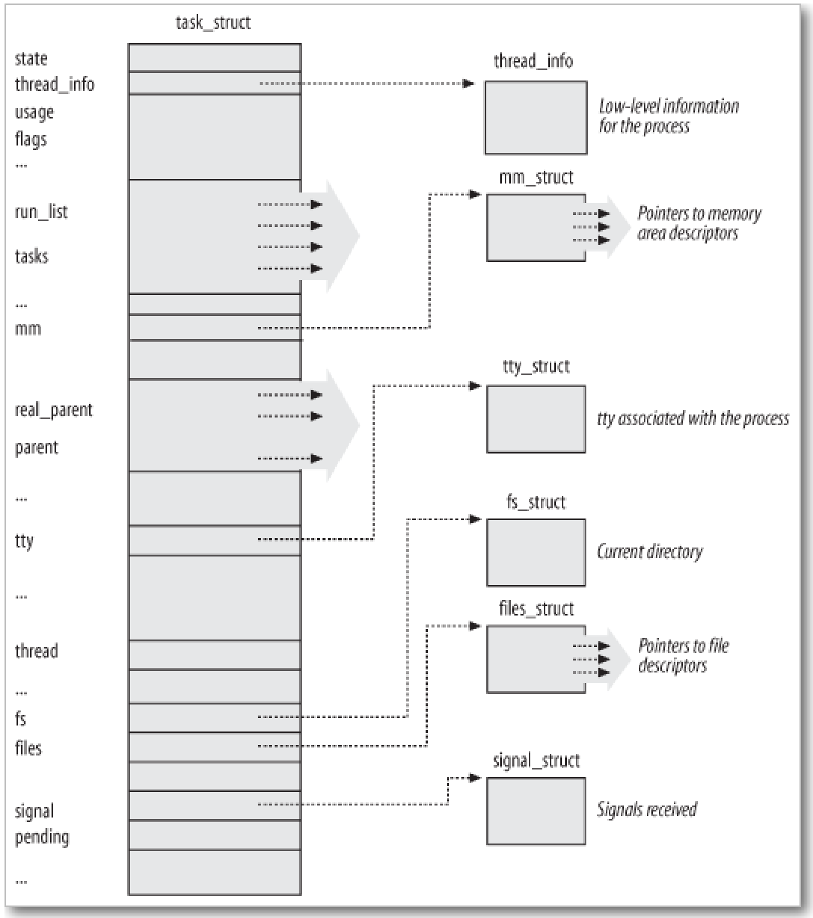
R: Runnable
S: Sleeping
T: Traced or Stopped
D: Uninterruptible Sleep
Z: Zombie
<: High-priority task
N: Low-priority task
s: Session leader
+: In the foreground process group
l: Multi-threaded

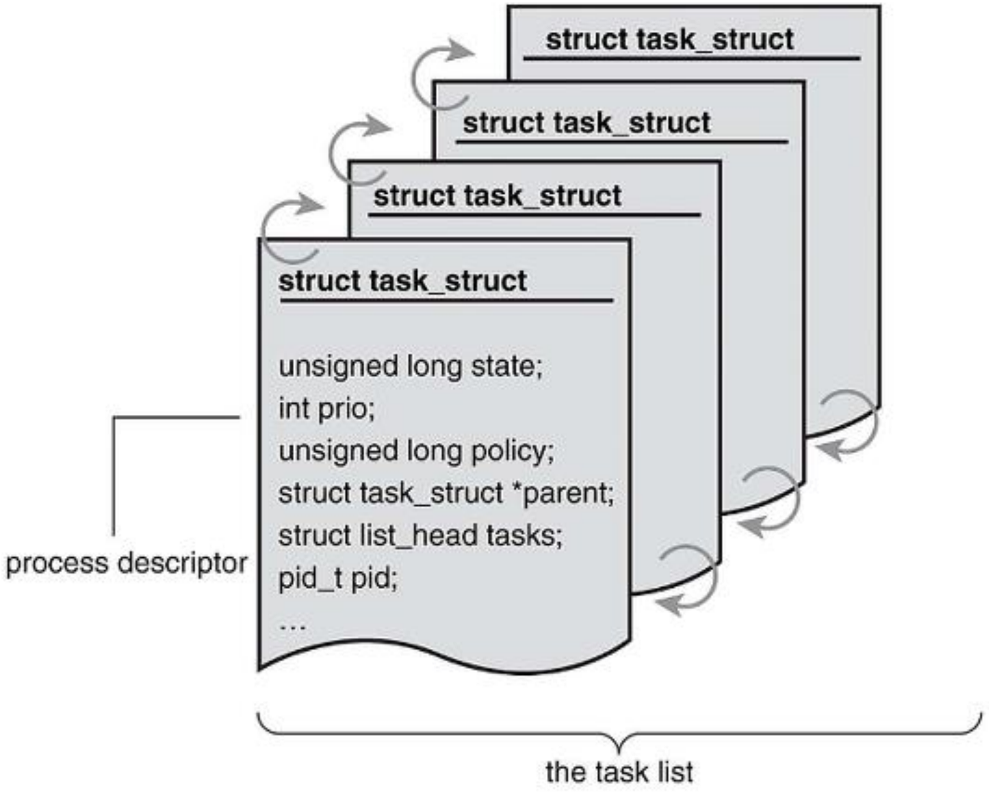
Process Data Structures

PCB (Process Control Block)

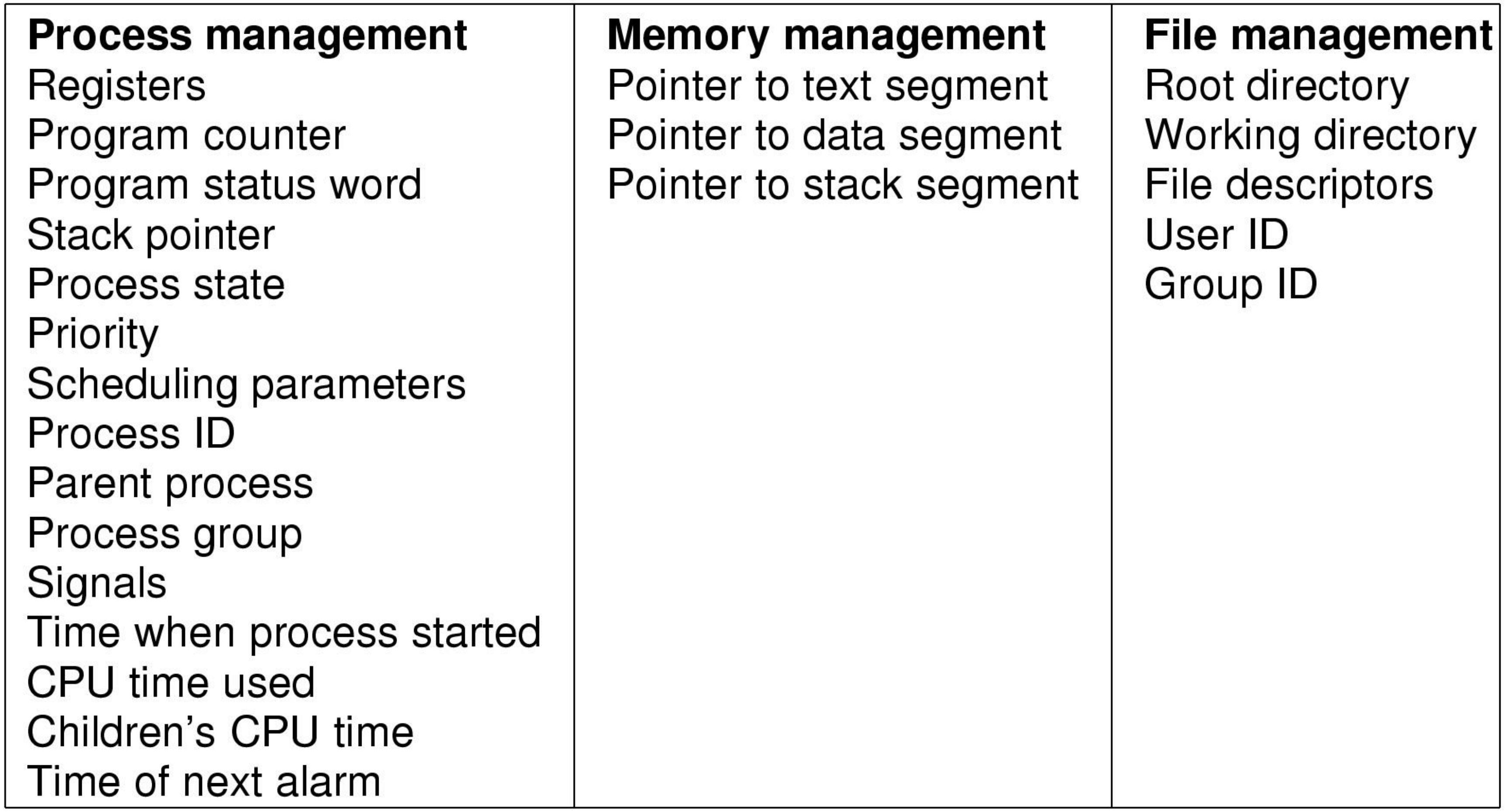
- Each PCB represents a process
- Contains all of the information about a process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory management information
  - Accounting information
  - I/O status information, etc.
- task_struct in Linux
  - 1456 bytes as of Linux 2.4.18
  - include/linux/sched.h

task_struct


task_struct
state
thread_info
usage
flags
...
run_list
tasks
...
mm
real_parent
parent
...
tty
...
thread
...
fs
files
signal
pending
...
thread_info
Low-level information for the process
mm_struct
Pointers to memory area descriptors
tty_struct
tty associated with the process
fs_struct
Current directory
files_struct
Pointers to file descriptors
signal_struct
Signals received


struct task_struct
struct task_struct
struct task_struct
struct task_struct
unsigned long state;
int prio;
unsigned long policy;
struct task_struct *parent;
struct list_head tasks;
pid_t pid;
…
process descriptor
the task list

Process Control Block (PCB)



Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	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		

PCBs and Hardware State

When a process is running:

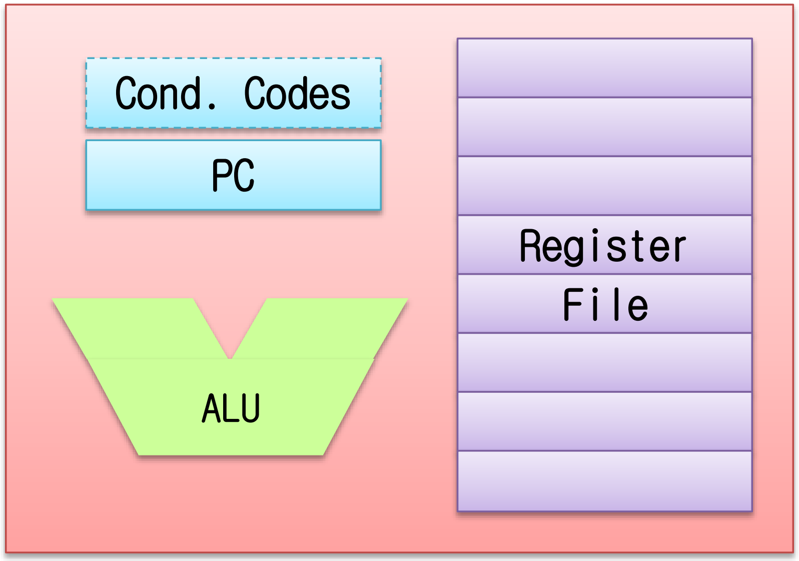
- Its hardware state is inside the CPU: PC, SP, registers

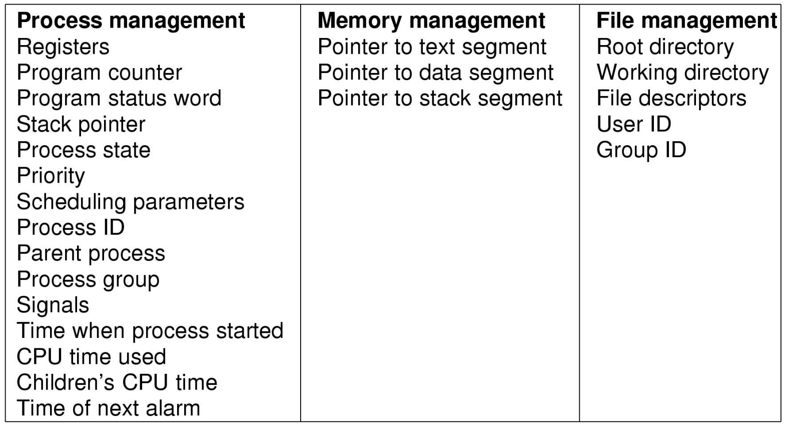
When the OS stops running a process:

- It saves the registers' values in the PCB

When the OS puts the process in the running state:

- It loads the hardware registers from the values in that process' PCB





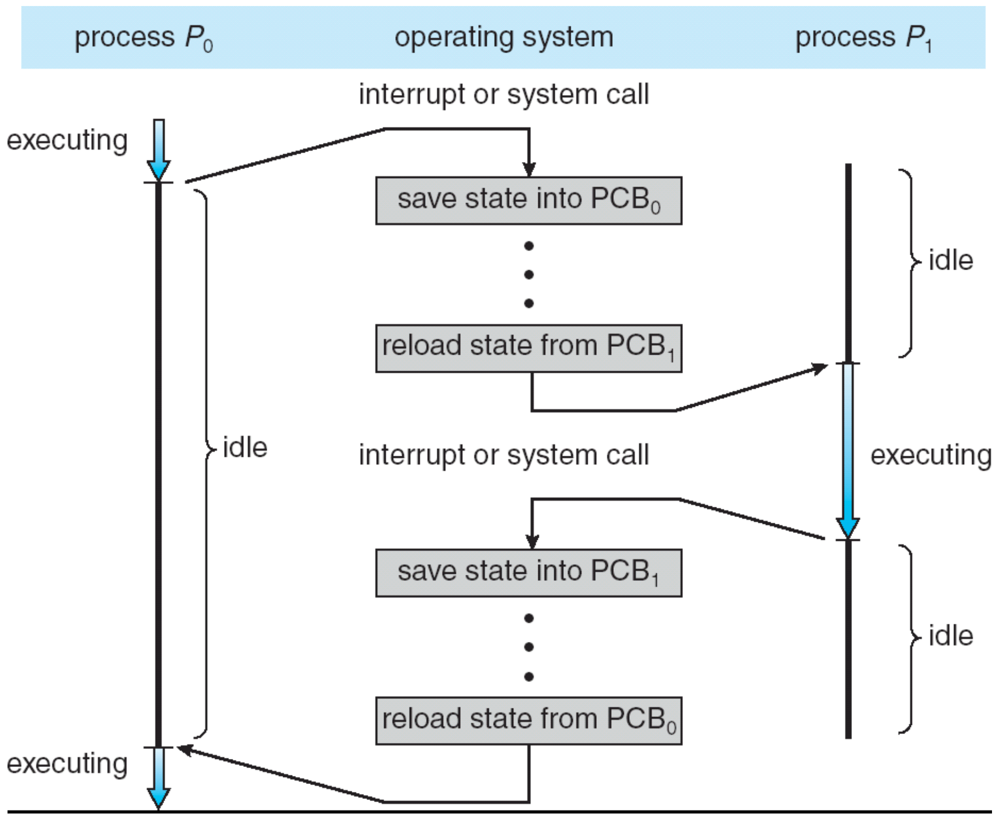
Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

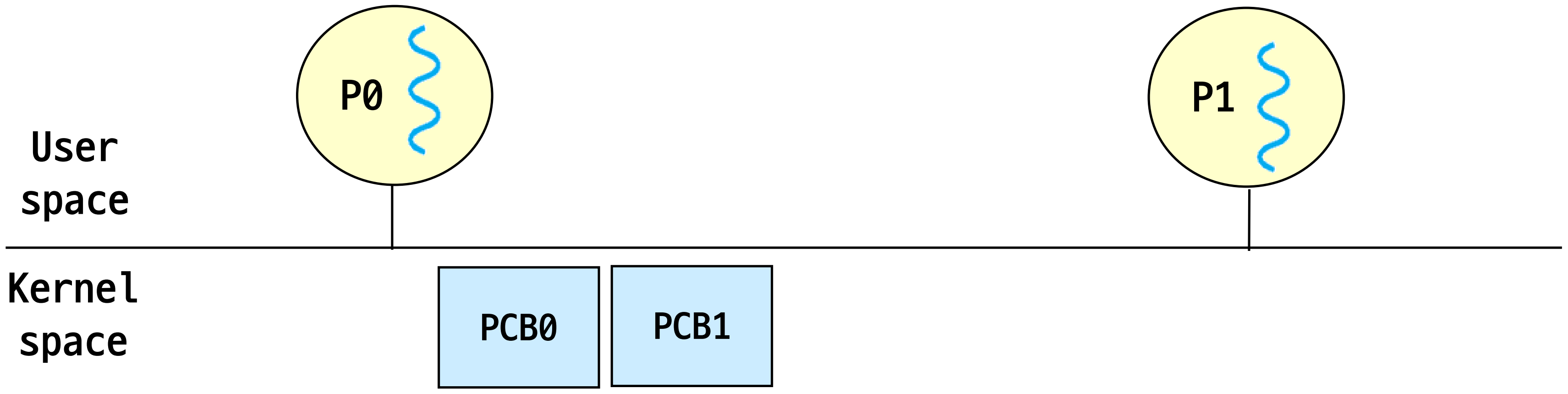
Context Switch (1)

Context switch (or process switch)

- The act of switching the CPU from one process to another
- Administrative overhead
  - Saving and loading registers and memory maps
  - Flushing and reloading the memory cache
  - Updating various tables and lists, etc.
- Context switch overhead is dependent on hardware support
  - Multiple register sets in UltraSPARC
  - Advanced memory management techniques may require extra data to be switched with each context
- 100s or 1000s of switches/s typically

Context Switch (2)

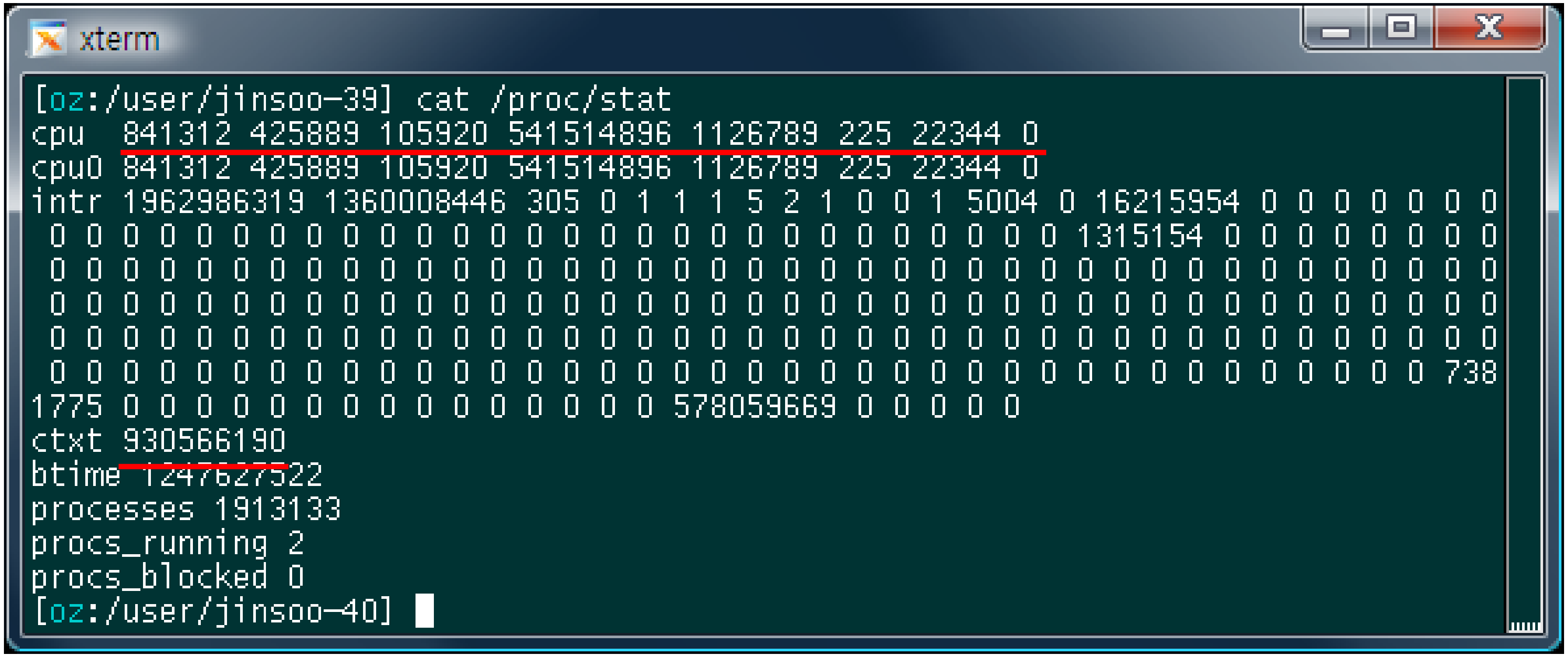




Context Switch (3)

Linux example

- Total 541,514,896 user ticks = 1511 hours = 63.0 days
- Total 930,566,190 context switches
- Roughly 86 context switches / sec (per CPU)



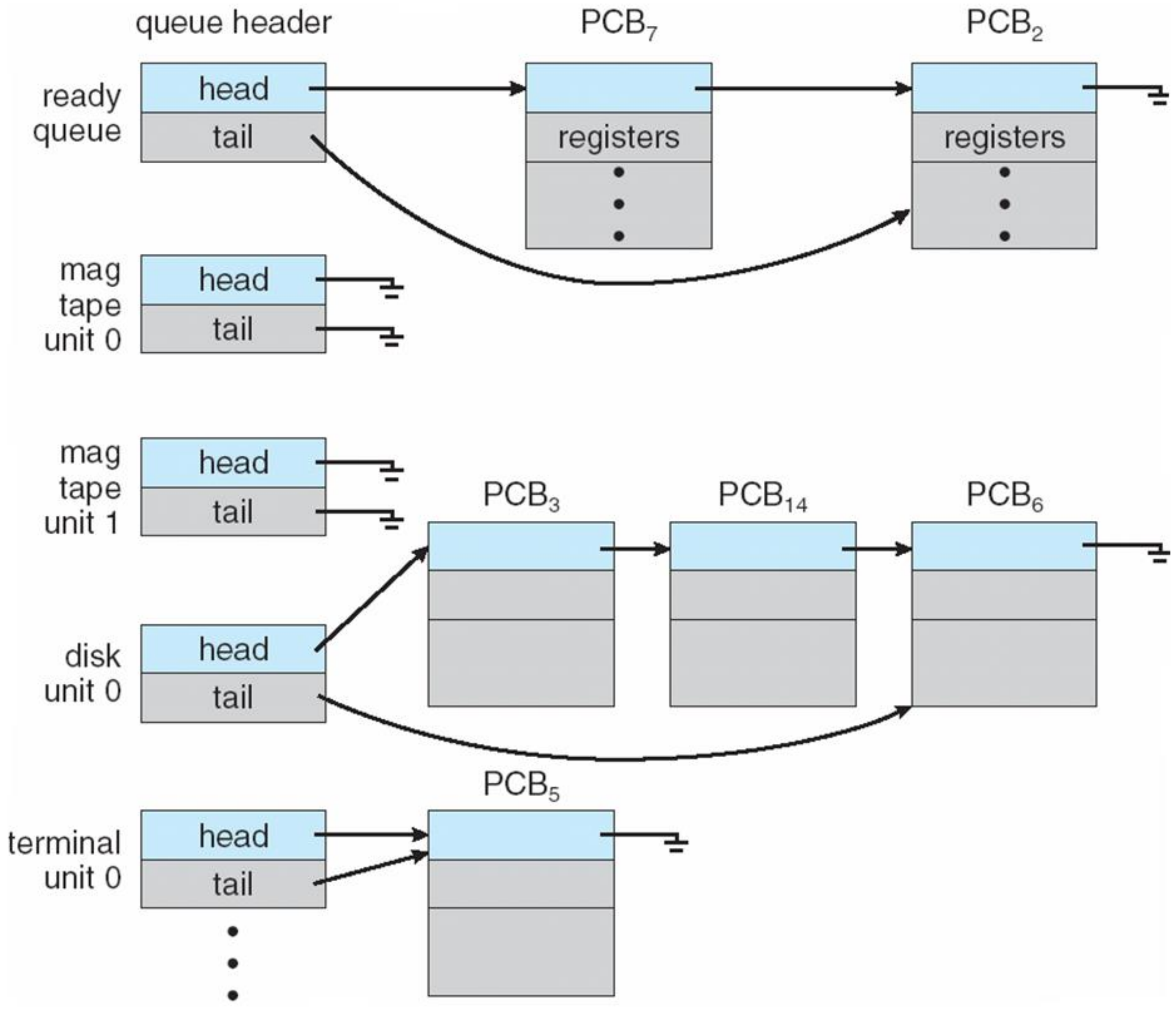
```
[oz:/user/jinsoo-39] cat /proc/stat
cpu  841312 425889 105920 541514896 1126789 225 22344 0
cpu0 841312 425889 105920 541514896 1126789 225 22344 0
intr 1962986319 1360008446 305 0 1 1 1 5 2 1 0 0 1 5004 0 16215954 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1315154 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 738
1775 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 578059669 0 0 0 0 0
ctxt 930566190
btime 1247627522
processes 1913133
procs_running 2
procs_blocked 0
[oz:/user/jinsoo-40]
```

Process State Queues (1)

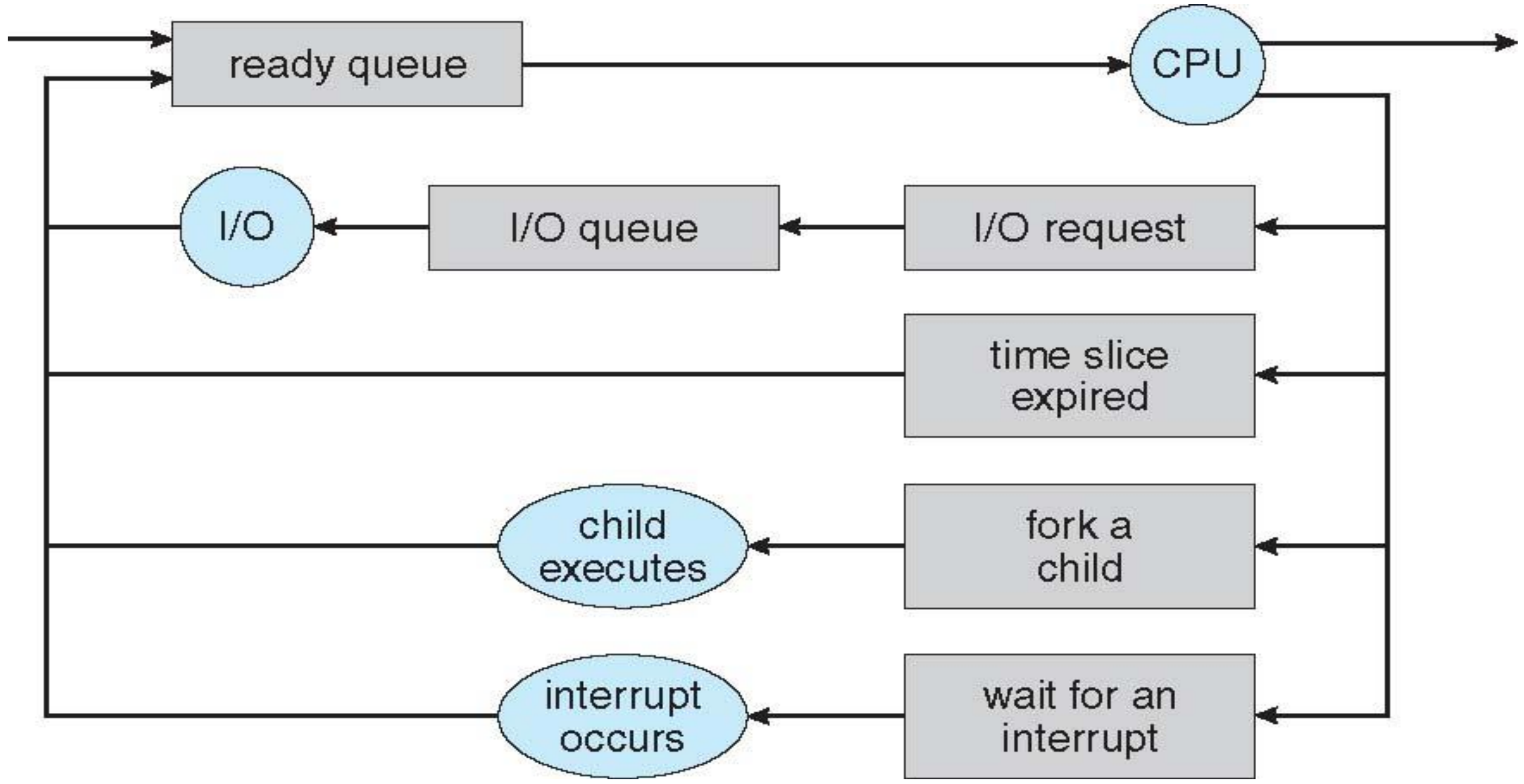
State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - Ready queue
  - Wait queue(s): there may be many wait queues, one for each type of wait (device, timer, message, …)
- Each PCB is queued onto a state queue according to its current state
- As a process changes state, its PCB is migrated between the various queues

Process State Queues (2)



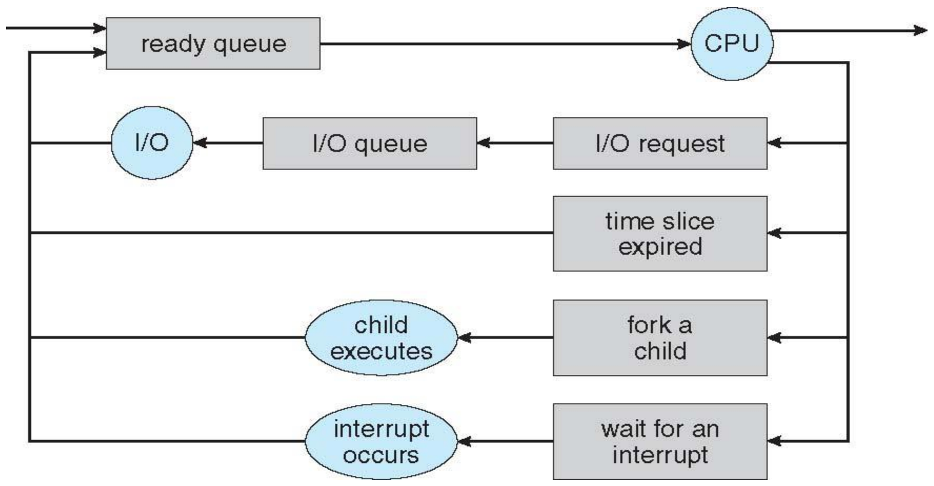
Process State Queues (3)



Process State Queues (4)

PCBs and state queues

- PCBs are data structures
  - Dynamically allocated inside OS memory
- When a process is created:
  - OS allocates a PCB for it
  - OS initializes PCB
  - OS puts PCB on the correct queue
- As a process computes:
  - OS moves its PCB from queue to queue
- When a process is terminated:
  - OS deallocates its PCB



Process Creation: UNIX (1)

`int fork()`

`fork()`

- Creates and initializes a new PCB
- Creates and initializes a new address space
- Initializes the address space with a copy of the entire contents of the address space of the parent
- Initializes the kernel resources to point to the resources used by parent (e.g., open files)
- Places the PCB on the ready queue
- Returns the child's PID to the parent, and zero to the child

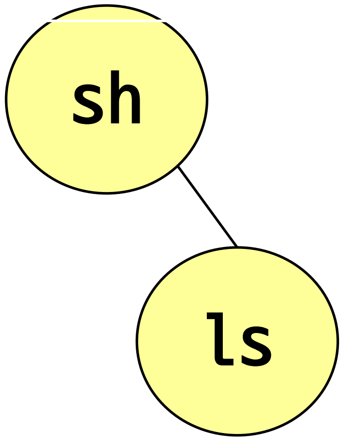
Process Creation: UNIX (2)

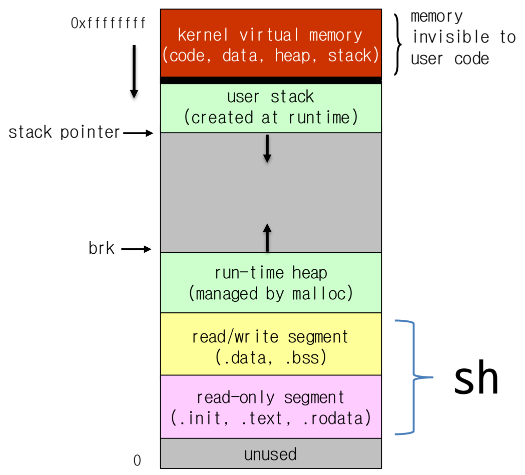
```
int exec (char *prog, char *argv[])
```

exec()

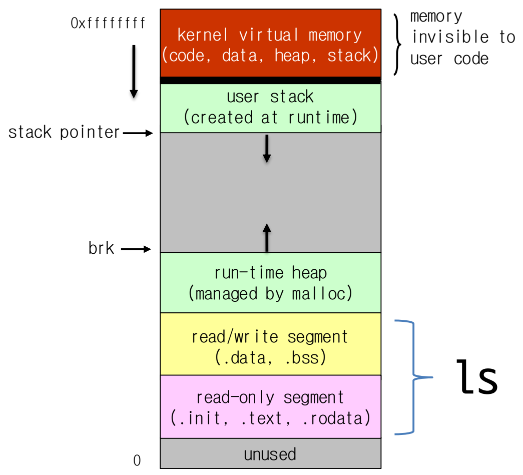
- Stops the current process
- Loads the program "prog" into the process' address space
- Initializes hardware context and args for the new program
- Places the PCB on the ready queue
  - Note: exec() does not create a new process

```
$ ls -aFl
```





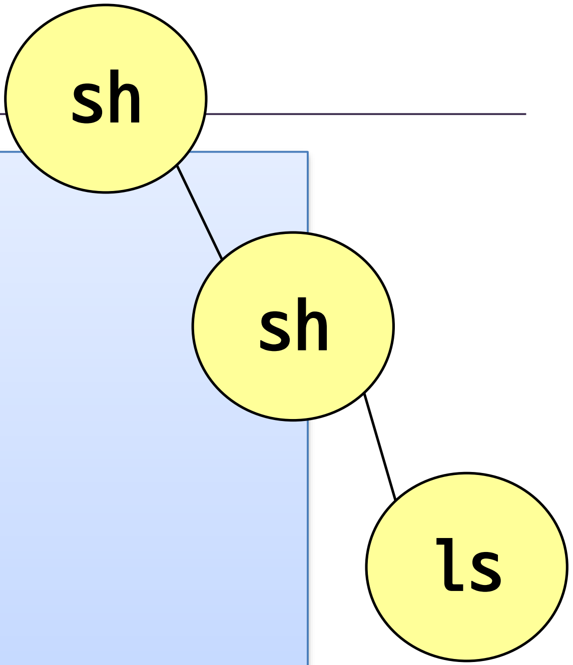
0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
user stack (created at runtime)
stack pointer
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
read-only segment (.init, .text, .rodata)
sh
unused
0



0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
user stack (created at runtime)
stack pointer
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
read-only segment (.init, .text, .rodata)
ls
unused
0

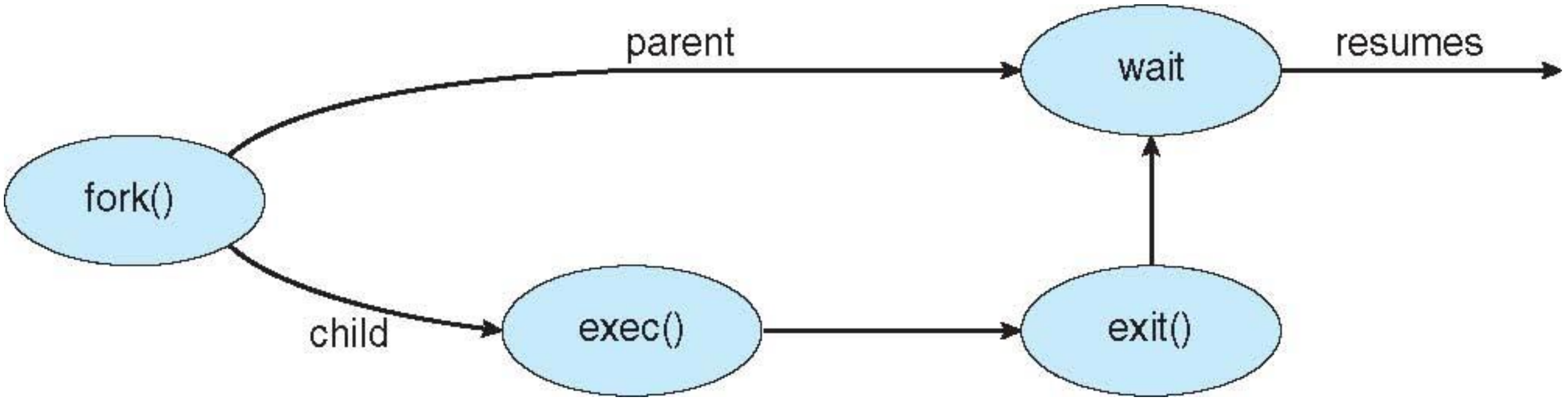
Simplified UNIX Shell

```
int main()
{
     while (1) {
          char *cmd = read_command();
          int pid;
          if ((pid = fork()) == 0) {
               /* Manipulate stdin/stdout/stderr for
                  pipes and redirections, etc. */
               exec(cmd);
               panic("exec failed!");
          } else {
               wait (pid);
          }
     }
}
```



```
$ ls -aF
./  ../  gitclone.sh*  temp3/  temp3.tgz  token/
$
```

Simplified UNIX Shell



```
int main()
{
    while (1) {
        char *cmd = read_command();
        int pid;
        if ((pid = fork()) == 0) {
            /* Manipulate stdin/stdout/stderr for
               pipes and redirections, etc. */
            exec(cmd);
            panic("exec failed!");
        } else {
            wait (pid);
        }
    }
}
```

```
$ ls -aF
./  ../  gitclone.sh*  temp3/  temp3.tgz  token/
$
```

Process Creation: NT

```
BOOL CreateProcess (char *prog, char *args, …)
```

CreateProcess()

- Creates and initializes a new PCB
- Creates and initializes a new address space
- Loads the program specified by "prog" into the address space
- Copies "args" into memory allocated in address space
- Initializes the hardware context to start execution at main
- Places the PCB on the ready queue

Why fork()?

Very useful when the child ...

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task
- Example: Web server

```
While (1) {
      int sock = accept();
      if ((pid = fork()) == 0) {
            /* Handle client request */
      } else {
            /* Close socket */
      }
}
```

Inter-Process Communications

Inside a machine

- pipe
- FIFO
- Shared memory
- Sockets

Across machines

- Sockets
- RPCs (Remote Procedure Calls)
- Java RMI (Remote Method Invocation)